cadence®

Xtensa® Debug Guide

For Xtensa NX and Xtensa LX Processors

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Issue Date: 10/2018

RI-2018.0

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

List of Tables

List of Figures

Preface

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- $ represents your shell prompt, in user-session examples.
- **`literal_input`** indicates literal command-line input.
- *`variable`* indicates a user parameter.
- `literal_keyword` (in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- *`... output ...`* indicates unspecified program output.
- *`[optional-variable]`* indicates an optional parameter.
- **`[`***`variable`***`]`** indicates a parameter within literal square-braces.
- **`{`***`variable`***`}`** indicates a parameter within literal curly-braces.
- **`(`***`variable`***`)`** indicates a parameter within literal parentheses.
- | means *OR*.
- *`(var1 | var2)`* indicates a required choice between one of multiple parameters.
- *`[var1 | var2]`* indicates an optional choice between one of multiple parameters.
- *`var1 [, varn]*`* indicates a list of 1 or more parameters (0 or more repetitions).
- $\overline{\texttt{SIGNALNAME}}$ indicates an active-low signal.

Terms

- *0x* at the beginning of a value indicates a hexadecimal value.
- *b* means bit.
- *B* means byte.
- *Mb* means megabit (1048576 bits).
- *MB* means megabyte (1048576 bytes).
- *PC* means program counter.
- *word* means 4 bytes.

Register Table Terms

- *RO* means read-only. The register or bits can be read. Writing has no effect, or is defined separately.
- *WO* means write-only. The register or bits can be written. Reading returns an undefined value, or is defined separately.
- *R/W* means read or write. The register or bits can be both read and written normally.
- *R/clr* means readable, clear on write. The register or bits can be read. When writing, bits written as ’1’ are cleared (set to 0), others are left unchanged.
- *R/set* means readable, set on write. The register or bits can be read. When writing, bits written as ’1’ are set (set to 1), others are left unchanged.

Changes from the Previous Version

1. Overview

This document describes the Xtensa architecture debug hardware (specifically, the Xtensa Debug module and the Traceport) and associated software.

Note: For information about the xt-gdb debugger, refer to the *Xtensa GNU Debugger User's Guide*.

1.1 Xtensa Debug Module

Debug hardware functionality in Xtensa processors is for the most part instantiated in the following two areas:

- the Debug option, which adds to the instruction set architecture (see the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for details)
- the Xtensa Debug module (also referred to in this document as the Debug module).

The Xtensa Debug module requires the Debug option.

The Xtensa Debug module is present when any of its following component parts are configured:

- On-Chip Debug (OCD) – see Chapter 5 through Chapter 7
- TRAX – see Chapter 10 through Chapter 15
- Performance Monitor – see Chapter 8 for Xtensa NX processors and Chapter 9 for Xtensa LX processors

The block diagram in Figure 1–1 depicts the Debug module (Debug module block) and its interfaces to other modules and agents. For reference, the diagram includes a number of non-Xtensa components (in this case from Arm; note that other systems are possible). The "Design Hierarchy" legend in Figure 1–1 shows the subsystem for each block.

Note that while most interfaces actually have signals that run in both directions, the diagram attempts to use directional arrows to show the master/slave relationship between two modules.

The OCD option allows an external agent (such as a debugger running on a host PC) to control the core for debugging purposes, such as accessing registers and memory, stopping, resuming, or stepping execution.

OCD has a sub-option, the Multiprocessor Support option (External Debug Interrupt option in the processor generator) that adds interrupt and status pins to the Xtensa processor. This option, also referred to as Break-In/Break-Out, simplifies the debugging of multiple-processor designs by enabling the synchronous halt and restart of selected processors.

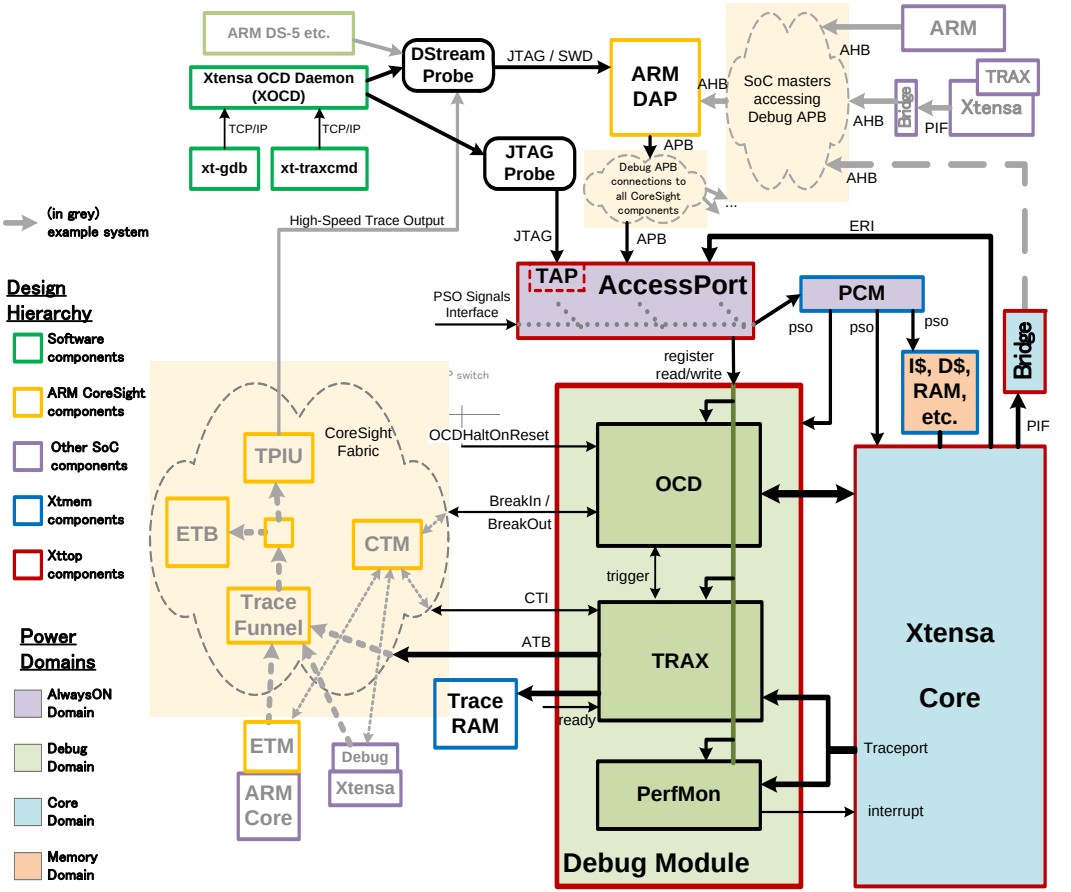


Figure 1–1. Debug Functionality

The Performance Monitor option (PerfMon) provides a set of hardware counters programmable to count key performance metrics.

The TRAX option implements hardware that collects compressed PC traces. This allows you to non-intrusively examine the execution stream of the running program.

In addition to the above components, the Xtensa Debug module has other associated components, some of which are optional:

- The Access Port (which allows access to debug functions via JTAG, APB, and ERI) — see Chapter 2
- Power, reset, and clock control — see Chapter 3
- CoreSight™-compatible registers and functionality — see Chapter 4

The Access Port implements three interfaces through which the Xtensa Debug module components are accessed: JTAG, APB, and ERI. All such accesses are 32-bit register reads and writes. These components are controlled and queried using 32-bit registers, in a common address space shared among all interfaces (except for JTAG, which uses 7-bit addressing (refer to Section 2.1).

The IEEE 1149.1 JTAG interface is always present. The associated Test Access Port (TAP) is part of the Access Port (see Section 2.3).

The optional APB interface is a slave memory addressable port compliant with the AMBA® 3.0 standard. This option is combined with the CoreSight-compatible registers and interfaces that make the Xtensa Debug module a CoreSight-compatible component. When TRAX is configured, this includes an ATB master interface that allows generated traces to be collected in a CoreSight trace system.

The external register interface (ERI) allows the Xtensa core to directly access the Xtensa Debug module using the external register access instructions, `RER` and `WER`. This provides software running on the core direct control of its own debug functions, such as through the TRAX control library documented in Chapter 13. Note that in the case of OCD, although the core can access certain OCD registers using this mechanism, software running on the core cannot debug itself that way: an external agent needs to manage the core when it is stopped using OCD. The word *external* in *external register interface* refers to the Xtensa Debug module being external to the core; at present, `RER` and `WER` accesses not serviced by the Xtensa Debug module are not routed to any separate external interface. See Section 2.5 for more details on ERI.

1.2 Traceport

The Xtensa processor Traceport (shown as an output interface from the Xtensa Core block in Figure 1–1) provides cycle-by-cycle information about the low-level execution state of the processor and the software running on it.

The Traceport is used internally by the Xtensa Debug module. It is also available as an external interface for system use.

Chapter 16 (for Xtensa NX processors) and Chapter 17 (for Xtensa LX processors) describe the Traceport and the software available to decode the sequences of values (traces) it generates.

1.3 Software Tools

Host debugging software (`xt-gdb`) accesses the Xtensa debug capabilities using the Xtensa OCD Daemon (XOCD in Figure 1–1). XOCD can drive various probes to connect to the AccessPort TAP, which in turn allows accessing the Debug module registers. Chapter 6 provides more details on XOCD and supported probes.

When the Xtensa Debug module is used within the Arm CoreSight debug infrastructure, XOCD supports the Arm DSTREAM probes only. This probe has included support for the Arm DAP, allowing XOCD to access an APB-attached Debug module without knowledge of the Arm DAP internals. Also, DSTREAM probes have native support for Arm Serial Wire Debug (SWD) interface to the Arm DAP. Currently, other probes cannot be used to connect to the Arm DAP and drive the Xtensa Debug module using the Debug APB interface.

The xt-traxcmd and xt-traxview tracing tools also use XOCD to access the TRAX and control the Xtensa Debug module program tracing capabilities. A more TRAX software-centric view of the diagram in Figure 1–1 is shown in Figure 10–30, which describes the software tracing tools in more detail.

Additional details on the Xtensa Debug module programming model are provided in Chapter 2.

1.4 Reference Documents

- *CoreSight Architecture Specification Version 1.0 ARM IHI 0029A 2004/9/29*
- *AMBA 3 APB Protocol Specification v1.0 Issue B 2004/8/17*

2. Debug Architecture

Debug logic (which includes the Debug module and Access Port) is operated through control and status registers that are accessible using three different interfaces — JTAG, APB, or ERI (the core's External Register Interface). For the most part, the Access Port implements the hardware that enables these interfaces.

Section 2.1 and Section 2.2 provide an architectural view of the registers in the Xtensa Debug module, including the Access Port. Section 2.3, Section 2.4, and Section 2.5 in turn describe each of the access interfaces: JTAG, APB, and ERI, respectively.

2.1 Debug Registers Overview

Debug module registers are read and written using a 32-bit memory-mapped interface directly accessible through APB and ERI interfaces, and accessible via an indirection scheme (based on the IEEE-ISTO Nexus 5000 standard) using the JTAG interface.

Each read or write transaction may be delayed by wait states or end in error. The busy and error reporting mechanisms are described in subsequent sections pertinent to JTAG, APB, or ERI.

2.1.1 Register Address Space

The Debug module registers are laid out in a 16 KB address space, as shown in Figure 2–2. Each of the three main functions of the Debug module—OCD, TRAX, and Performance Monitor—is allotted its own 4K page.

The register layout is designed to be CoreSight-compatible. It includes CoreSight Component Management registers at the expected location at the end of its address space. These describe the component (such as the size of its address space, which must be a power-of-2 multiple of 4K pages) and include specific control registers. The most significant bit (bit 31) of the APB address is used to distinguish accesses by processors within the SoC (bit 31 = 0) from those made by an external debug agent (bit 31 = 1). Thus the Debug module can act as a CoreSight component, accessed over a Debug APB bus, for example from an ARM Debug Access Port (DAP). See Chapter 4 for CoreSight component details, and Section 4.3 for specific details about locking accesses by processors within the SoC (bit31 = 0).

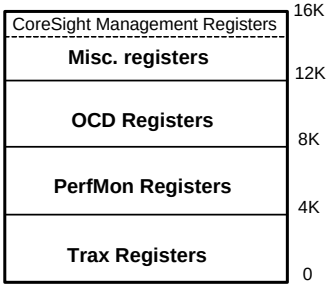


Figure 2–2. Debug Module Address Map

2.2 Detailed Address Map

Table 2–1 shows the overall register space allocation across all interfaces.

Table 2–1. Overview Memory Map of Debug Module

APB Address	ERI Address	Nexus Reg Addr	Description
0x0000 – 0x0FFF	0x100000 – 0x100FFF	0x00 – 0x1F	TRAX registers
0x1000 – 0x1FFF	0x101000 – 0x101FFF	0x20 – 0x3F	Performance Monitor registers
0x2000 – 0x2FFF	0x102000 – 0x102FFF	0x40 – 0x4F	OCD registers
–	–	0x50 – 0x57	Reserved
0x3000 – 0x3EFF	0x103000 – 0x103EFF	0x58 – 0x5F	Miscellaneous registers
0x3F00 – 0x3FFF	0x103F00 – 0x103FFF	0x60 – 0x7F	CoreSight registers

Note that the APB and ERI addresses match, except for an offset:

ERI address = 0x100000 + APB address.

The APB address space starts at zero because it is effectively just an offset into the accessible 16 KB space. As an APB slave, the Xtensa core will be placed in an APB address space(s) at some 16 KB aligned address, along with other devices. Decoding of these higher address bits is done outside the Xtensa core. APB address bit 31 is also managed by the system decoding for distinguishing system-external vs. system-internal accesses as defined in Section 4.3.

The ERI addresses, on the other hand, are used with `WER` and `RER` instructions on the core.

The JTAG NAR addresses differ more significantly from the other two. They are organized much more compactly due to the limited 7-bit addressing of NAR. See Section 2.3.5 for details.

Note that while APB address bit 31 is used to distinguish system-internal vs. system-external APB accesses, the same does not apply to JTAG or ERI. The distinction is needed to properly implement locking for APB accesses, but neither JTAG nor ERI is subject to locking.

For Xtensa NX processors: Table 2–2 lists all Debug module registers, with sub-heading rows indicating the associated subsystem (*OCD*, *TRAX*, *Performance Monitor*, *Miscellaneous* or *CoreSight*) and related chapter or section. Addresses not shown in this table are reserved.

Table 2–2. Xtensa NX Processor Memory Map of All Debug Module Registers

APB Address (ERI Offset)	Nexus Reg Addr	Name	Description
TRAX Registers (Chapter 15)			
0x0000	0x00	TRAXID	ID register (the same as `OCDID`)
0x0004	0x01	TRAXCTRL	Control register
0x0008	0x02	TRAXSTAT	Status register
0x000C	0x03	TRAXDATA	Data register
0x0010	0x04	TRAXADDR	Address register
0x0014	0x05	TRIGGERPC	Stop PC
0x0018	0x06	PCMATCHCTRL	Stop PC Range
0x001C	0x07	DELAYCNT	Post Stop Trigger Capture Size
0x0020	0x08	MEMADDRSTART	Trace Memory Start Address
0x0024	0x09	MEMADDREND	Trace Memory End Address
0x0040	0x10	EXTTIMELO	External Time Low
0x0044	0x11	EXTTIMEHI	External Time High
Performance Monitor Registers (Chapter 8)			
0x1000	0x20	PMG	Performance counters control register
0x1010	0x24	INTPC	PC at cycle of last event that caused interrupt
0x1080 - 0x109C	0x28 - 0x2F	PM0 - PM7	Performance counter values
0x1100 - 0x111C	0x30 - 0x37	PMCTRL0 - PMCTRL7	Performance counter control registers
0x1180 - 0x119C	0x38 - 0x3F	PMSTAT0 - PMSTAT7	Performance counter status registers
OCD Registers (Chapter 5)			
0x2000	0x40	OCDID	ID Register (the same as `TRAXID`)
0x2008	0x42	DCRCLR	Debug Control Register (write to clear)
0x200C	0x43	DCRSET	Debug Control Register (write to set)
0x2010	0x44	DSR	Debug Status Register
0x2014	0x45	DDR	Debug Data Register; for host to/from target transfers

Table 2–2. Xtensa NX Processor Memory Map of All Debug Module Registers

APB Address (ERI Offset)	Nexus Reg Addr	Name	Description
0x2018	0x46	DDREXEC	Alias to DDR; executes DIR when accessed
0x201C	0x47	DIR0EXEC	Alias to DIR0; executes the instruction when written
0x2020	0x48	DIR0	Debug Instruction Register (first 32 bits)
0x2024 - 0x203C	0x49 - 0x4F	DIR1 - DIR7	Debug Instruction Register (other bits)
Miscellaneous Registers (Chapter 3 and Section 2.2.3)			
0x3020		PWRCTL	Power and reset control (resides in Access Port)
0x3024		PWRSTAT	Power and reset status (resides in Access Port)
0x3028	0x5A	ERISTAT	ERI transaction status
0x302C	0x5B	FAULTINFOLO	Fault information register low portion
0x3030	0x5C	FAULTINFOHI	Fault information register high portion
0x3034	--	AXI_ECC_ENABLE	AXI bus ECC enable
0x3038	--	AXI_ECC_STATUS	AXI bus ECC status
CoreSight Registers (Chapter 4)			
0x3F00	0x60	ITCTRL	Integration Mode Control Register
0x3FA0	0x68	CLAIMSET	Claim Tag Set Register
0x3FA4	0x69	CLAIMCLR	Claim Tag Clear Register
0x3FB0	0x6C	LOCKACCESS	Writing 0xC5ACCE55 unlocks internal master access
0x3FB4	0x6D	LOCKSTATUS	Current locking status
0x3FB8	0x6E	AUTHSTATUS	Authentication Status
0x3FC8	0x72	DEVID	Device ID, including endianness
0x3FCC	0x73	DEVTYPE	Device type
0x3FD0	0x74	Peripheral ID4	Peripheral ID, including component page size (16 KB)
0x3FD4	0x75	Peripheral ID5	Reserved
0x3FD8	0x76	Peripheral ID6	Reserved
0x3FDC	0x77	Peripheral ID7	Reserved
0x3FE0	0x78	Peripheral ID0	Peripheral ID, including part number
0x3FE4	0x79	Peripheral ID1	Peripheral ID, including part number and JEDEC code
0x3FE8	0x7A	Peripheral ID2	Peripheral ID, including revision and JEDEC code
0x3FEC	0x7B	Peripheral ID3	Peripheral ID, including mask revision
0x3FF0	0x7C	Component ID0	Preamble
0x3FF4	0x7D	Component ID1	Component ID, incl. component class (CoreSight)
0x3FF8	0x7E	Component ID2	Preamble
0x3FFC	0x7F	Component ID3	Preamble

For Xtensa LX processors: Table 2–3 lists all Debug module registers, with sub-heading rows indicating the associated subsystem (*OCD*, *TRAX*, *Performance Monitor*, *Miscellaneous* or *CoreSight*) and related chapter or section. Addresses not shown in this table are reserved.

Table 2–3. Xtensa LX Processor Memory Map of All Debug Module Registers

APB Address (ERI Offset)	Nexus Reg Addr	Name	Description
TRAX Registers (Chapter 14)			
0x0000	0x00	TRAXID	ID register (the same as `OCDID`)
0x0004	0x01	TRAXCTRL	Control register
0x0008	0x02	TRAXSTAT	Status register
0x000C	0x03	TRAXDATA	Data register
0x0010	0x04	TRAXADDR	Address register
0x0014	0x05	TRIGGERPC	Stop PC
0x0018	0x06	PCMATCHCTRL	Stop PC Range
0x001C	0x07	DELAYCNT	Post Stop Trigger Capture Size
0x0020	0x08	MEMADDRSTART	Trace Memory Start Address
0x0024	0x09	MEMADDREND	Trace Memory End Address
0x0040	0x10	EXTTIMELO	External Time Low
0x0044	0x11	EXTTIMEHI	External Time High
Performance Monitor Registers (Chapter 8)			
0x1000	0x20	PMG	Performance counters control register
0x1010	0x24	INTPC	PC at cycle of last event that caused interrupt
0x1080 - 0x109C	0x28 - 0x2F	PM0 - PM7	Performance counter values
0x1100 - 0x111C	0x30 - 0x37	PMCTRL0 - PMCTRL7	Performance counter control registers
0x1180 - 0x119C	0x38 - 0x3F	PMSTAT0 - PMSTAT7	Performance counter status registers
OCD Registers (Chapter 5)			
0x2000	0x40	OCDID	ID Register (the same as `TRAXID`)
0x2008	0x42	DCRCLR	Debug Control Register (write to clear)
0x200C	0x43	DCRSET	Debug Control Register (write to set)
0x2010	0x44	DSR	Debug Status Register
0x2014	0x45	DDR	Debug Data Register; for host to/from target transfers
0x2018	0x46	DDREXEC	Alias to `DDR`; executes `DIR` when accessed
0x201C	0x47	DIR0EXEC	Alias to DIR0; executes the instruction when written
0x2020	0x48	DIR0	Debug Instruction Register (first 32 bits)
0x2024 - 0x203C	0x49 - 0x4F	DIR1 - DIR7	Debug Instruction Register (other bits)
Miscellaneous Registers (Chapter 3 and Section 2.2.3)			
0x3020		PWRCTL	Power and reset control (resides in Access Port)
0x3024		PWRSTAT	Power and reset status (resides in Access Port)

APB Address (ERI Offset)	Nexus Reg Addr	Name	Description
0x3028	0x5A	ERISTAT	ERI transaction status
CoreSight Registers (Chapter 4)			
0x3F00	0x60	ITCTRL	Integration Mode Control Register
0x3FA0	0x68	CLAIMSET	Claim Tag Set Register
0x3FA4	0x69	CLAIMCLR	Claim Tag Clear Register
0x3FB0	0x6C	LOCKACCESS	Writing 0xC5ACCE55 unlocks internal master access
0x3FB4	0x6D	LOCKSTATUS	Current locking status
0x3FB8	0x6E	AUTHSTATUS	Authentication Status
0x3FC8	0x72	DEVID	Device ID, including endianness
0x3FCC	0x73	DEVTYPE	Device type
0x3FD0	0x74	Peripheral ID4	Peripheral ID, including component page size (16 KB)
0x3FD4	0x75	Peripheral ID5	Reserved
0x3FD8	0x76	Peripheral ID6	Reserved
0x3FDC	0x77	Peripheral ID7	Reserved
0x3FE0	0x78	Peripheral ID0	Peripheral ID, including part number
0x3FE4	0x79	Peripheral ID1	Peripheral ID, including part number and JEDEC code
0x3FE8	0x7A	Peripheral ID2	Peripheral ID, including revision and JEDEC code
0x3FEC	0x7B	Peripheral ID3	Peripheral ID, including mask revision
0x3FF0	0x7C	Component ID0	Preamble
0x3FF4	0x7D	Component ID1	Component ID, incl. component class (CoreSight)
0x3FF8	0x7E	Component ID2	Preamble
0x3FFC	0x7F	Component ID3	Preamble

2.2.1 Access Port Registers

Not all the registers are implemented in the Debug module. The reset and power control and status registers reside physically in the Access Port. This is because the Debug module is reset from these registers, and is shut off as part of the Debug power domain, so these registers need to be accessible independently of the Debug module per se. The Debug module must be powered on in order for most registers to be accessible.

Access Port registers are common to all access interfaces in the Debug module. These registers are listed in Table 2–2, under the Miscellaneous modules (see Module column), i.e. address range 0x3000 to 0x3BFF. Subsections that follow discuss each in more detail.

2.2.2 Debug Module ID Register (OCDID and TRAXID)

The ID register can be accessed using either `OCDID` register or the `TRAXID` register. This allows external software to access information about Debug module from within the components it actually uses (e.g. the debugger might use OCD but not TRAX) or while one of the components might not even exist (e.g. TRAX).

Figure 2–3 shows the information contained in the ID register.

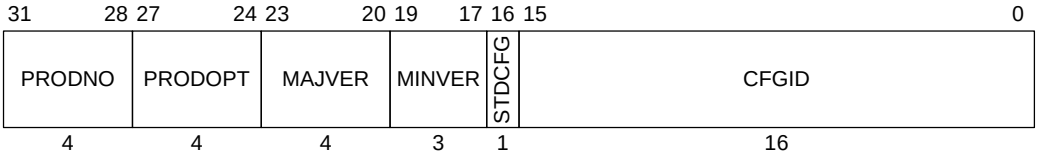


Figure 2–3. ID Register Generic Layout

The values of the fields within the ID register are release-specific and are reserved for Cadence use.

2.2.3 ERISTAT Register

This register has only a single-bit field, which is used to convey the result of ERI write transactions. Refer to Section 2.5 for more detail. This register is listed in Table 2–2, under the Miscellaneous modules.

Table 2–4. ERISTAT Register

Field	Width	Bits	Description
-	31	31:1	Reserved
WRSUC	1	0	ERI write success indication

2.3 JTAG Interface

This section describes the implementation of the JTAG interface in the Access Port. The portion of the Access Port that deals with this interface is known as the TAP (Test Access Port) Controller and is described in the following sections.

2.3.1 TAP Standard

This section summarizes the TAP standard. Complete details of the JTAG TAP standard are outside the scope of this document. The interested reader is referred to the *IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE Std. 1149.1a-1993*.

A TAP interface consists of five signals: TCK, TMS, TDI, TDO, and $\overline{\text{TRST}}$. The TAP controller is a synchronous finite state machine controlled by the TCK, TMS and $\overline{\text{TRST}}$ signals of the TAP. Figure 2–4 describes the state machine. The value of TMS on every rising edge of TCK (as shown for each transition in Figure 2–4) determines transitions of the state machine. Also, holding $\overline{\text{TRST}}$ low causes an asynchronous transition to the Test-Logic-Reset state regardless of the state of TCK and TMS.

Conceptually, the test logic accessed via the TAP consists of an Instruction Register (IR) and several Data Registers (DR). Each such register generally exists in two adjacent forms: as a serial shift register, and as a parallel register. When data is shifted serially, exactly one of these registers is selected, and its serial shift register connected to the TDI and TDO signals of the TAP. TDI is connected to the MSB of the shift register, and TDO to the LSB. Hence, data is always shifted in and out with the least-significant bit first.

Access to a TAP register proceeds as follows, using TMS and TCK transitions to advance through the various TAP controller states (as shown in Figure 2–4).

- Either the instruction register or one of the data registers is selected by moving to the Select-IR or Select-DR state, respectively. In the case of data registers, which data register is selected is a function of the current instruction register contents (for example, see Table 2–6 for the default Xtensa TAP).
- In the Capture state, generally some value is transferred to the serial shift register. Usually this is from a corresponding parallel register. However the instruction serial shift register does not get a copy of the current instruction. Instead, it is given a value whose least significant bit is 1 and second least significant bit is 0 (that is, whose binary value ends in 01). Also, the one-bit bypass serial shift data register has no parallel equivalent, and is set to zero in this state.
- In the Shift state, TDO is active and the selected shift register is shifted towards its LSB by one bit (shifting in its most significant bit from TDI, and shifting out its least significant bit to TDO). The desired number of bits (usually the number of bits in the selected register) are shifted by entering the Shift state that many times. This is normally done by keeping TMS low for the appropriate number of rising edges of TCK.
- In the Update state (most commonly reached from the Exit1 state), the serial shift register is usually copied to its corresponding parallel register (if there is one).

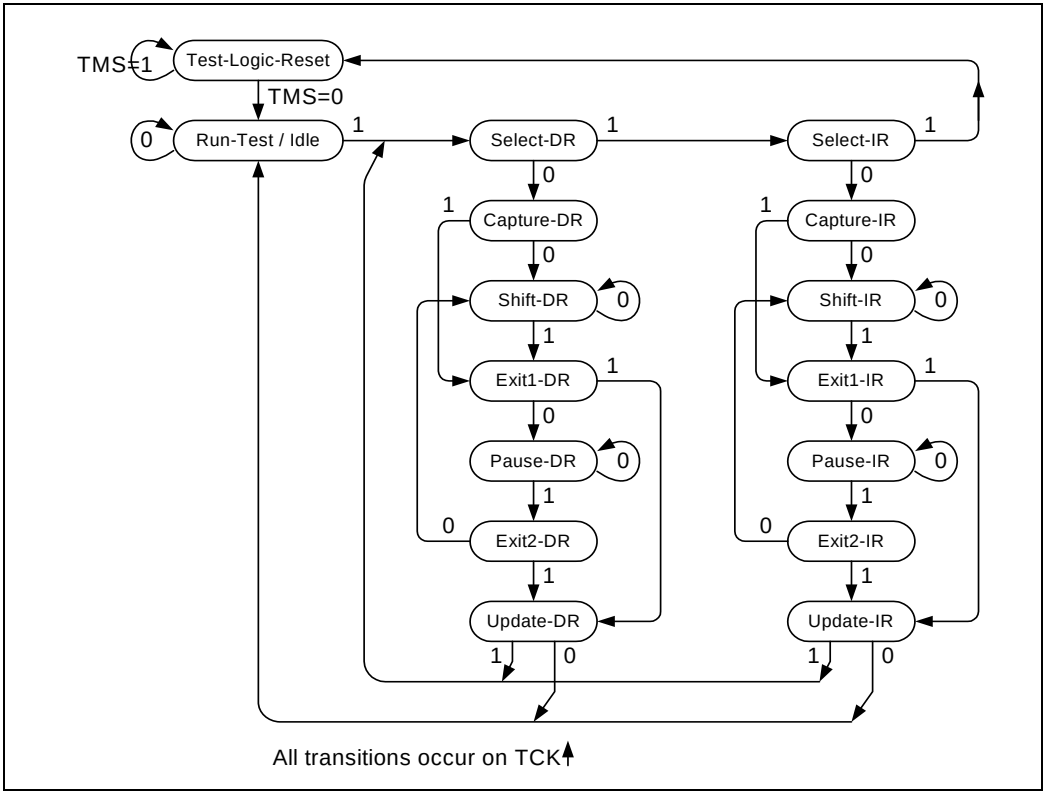


Figure 2–4. TAP Controller State Machine

In actual operation, you would typically first scan in the instruction register, which has the effect of selecting one of the available data registers. Loading the instruction register may also have other instruction-specific side-effects. Then you would scan in/out the selected data register.

All shift registers have a defined, fixed length. All data is shifted in and out starting with the least-significant-bit (that is, LSB-first). This is defined by the TAP standard and is independent of Xtensa core endianness.

By design, the Test-Logic-Reset state can be reached from any other TAP controller state in at most five rising edges of TCK where TMS equals 1.

TMS should be held high on the rising edge of $\overline{\text{TRST}}$ (that is, when releasing $\overline{\text{TRST}}$) to guarantee deterministic results. Otherwise the Run-Test/Idle state might be entered immediately.

The Exit1-xx, Pause-xx, and Exit2-xx states allow pausing serial shifting while maintaining TCK active. This may be useful in some test environments. For a TAP that is used solely for Xtensa core debugging, these states are not particularly useful, but are implemented nonetheless according to standard.

Note: Multiple reads of Xtensa data registers by traversing only the data branch of the TAP state machine will not return the latest value of the data registers. The TAP instruction needs to be re-executed.

2.3.2 TAP Signals

Table 2–5 describes five standard TAP signals as implemented in the TAP controller provided by Cadence. Relative to the standard, it adds a separate tri-state enable signal for TDO (see Section 2.3.2.2), and always includes $\overline{\text{TRST}}$.

Table 2–5. Cadence TAP Controller Signals

Standard Name	RTL Name	Direction	Description
TCK	JTCK	Input	Clock that controls sampling of TMS, TDI and update of TDO.
TMS	JTMS	Input	Input to TAP controller state machine.
$\overline{\text{TRST}}$	JTRST	Input	Active low reset input for asynchronous initialization of the TAP controller.
TDI	JTDI	Input	Selected serial instruction/data shift register input.
TDO	JTDO	Output	Selected serial instruction/data shift register tri-state output.
	JTDOEn	Output	Selected serial instruction/data shift register tri-state enable.

2.3.2.1 Input Signal Requirements

The IEEE 1149.1 specification requires that TMS, TDI, and $\overline{\text{TRST}}$ shall be such that an undriven input produces a logical response identical to the application of a logic 1. The system designer is responsible for ensuring this.

2.3.2.2 Output Signal Requirements

The IEEE 1149.1 specification defines TDO as a tri-state output signal that is enabled only during serial output in the TAP controller Shift states (see Figure 2–4 and Table 2–6). Xtensa's TAP controller drives JTDO and JTDOEn signals as ordinary logic outputs; and the system designer must provide the actual tri-state buffer (usually at the chip boundary).

Note: JTDOEn is an active-high output.

2.3.3 TAP Instruction Registers (IRs)

Available TAP instructions are listed in Table 2–6. The width of the IR is 5 bits. Encodings not shown in this table are reserved for future use; until implemented otherwise, they all select the 1-bit bypass data register. Table 2–6 also shows the TAP data registers implemented in the TAP controller.

Table 2–6. TAP Instructions

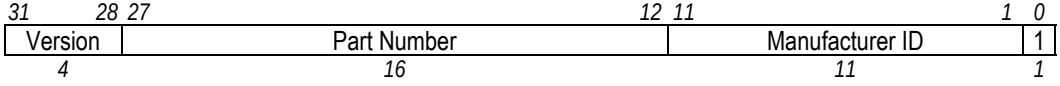
Instruction	Encoding	Selected TAP Data Reg	Length	Description
-	00xxx	-	-	(*reserved*)
PWRCTL	01000	PWRCTRL	8	Power and Reset Control register
PWRSTAT	01001	PWRSTAT	8	Power and Reset Status register
-	01010-11011	-	-	(*reserved*)
NARSEL	11100	NAR then NDR	8 / 32	Alternating Nexus Addr/Data Regs. Starts with NAR
IDCODE	11110	device id	32	IEEE 1149.1 optional instruction
BYPASS	11111	bypass	1	IEEE 1149.1 required instruction

Note: In the Capture-IR state, the instruction serial shift register is always loaded with 00001; not with the previous instruction. (The TAP standard requires the loaded value to be xxx01, where xxx can be any data, but must be a constant all 0's or all 1's if unused)

The following standard TAP instructions are not implemented, as they are not required for TAP operation: EXTEST, SAMPLE/PRELOAD, RUNBIST, HIGHZ, and CLAMP.

2.3.4 Device ID (IDCODE) Register in Xtensa LX and Xtensa NX Processors

IDCODE is a 32-bit TAP data register containing a constant value identifying the device managed by this TAP. The IEEE JTAG standard defines a 32-bit IDCODE composed of 4 fields:



Bit 0 is a marker bit, and is always 1.

The 11-bit Manufacturer ID is the JEDEC Manufacturer ID code that corresponds to the particular company/implementation. The JEDEC Manufacturer ID for Xtensa processors is: 01001110010 (0x272) which is JEDEC ID 114 (0x72) bank 5. If you include the Marker, this makes the lower 12 bits of IDCODE to be 0x4E5. See Section 4.7.1.2 for how those bits get encoded in CoreSight registers.

For Xtensa LX processors, upper bits are fixed except for bit 20 which carries the information about the core endianness. As a result, the Xtensa LX processor's 32-bit IDCODE is 0x121034E5 for big endian and 0x120034e5 for little endian cores.

For Xtensa NX processors, the IDCODE introduces a 10-bit version field (CIDVers) at IDCODE[25:16], which uniquely identifies the hardware release of the Xtensa processor. In addition, the version field has a value of 0x2, and IDCODE[15:12] is fixed at 0x3.Synchronization Between TAP and Processor

The TAP clock (JTCK) and the Xtensa core clock (CLK) are entirely asynchronous. No phase or frequency relationship exists between them, except that JTCK must be slower than the core clock. That is, for every edge of JTCK there must be at least three edges of the core clock, assuming a nominally 50% duty cycle for either clock. This means that JTCK must be less than one third of the core clock frequency. Because JTCK is always slower than the core clock - and always slow enough - a transition in a signal driven from a flip/flop in the JTCK domain will be effective in the core clock domain before the next JTCK cycle.

Because of duty cycle and phase/frequency variations in either clock, the effective maximum JTCK frequency is somewhat lower than the one third required. Cadence recommends a ratio of **one fourth** to better accommodate imperfect clock conditions. This assumption is carried through the OCD daemon software documentation of Chapter 6.

The core clock is not slowed down or synchronized in any way when the core is *Stopped* using OCD. Input signals from the TAP that are used in core clock domain logic are synchronized before they are used. See the following subsection for an example.

2.3.4.1 TAP Data Register Synchronization

Each TAP data register has a shift (or serial) and update (or parallel) register. The shift registers are built using boundary-scan cells that are typical of TAP test data registers and are clocked using JTAG JTCK. However, because the parallel register works in the Xtensa core clock domain, there are two registers (capture and parallel) instead of one. The capture register is used for synchronization between the JTCK (TAP) and core clock domains as illustrated in Figure 2–5.

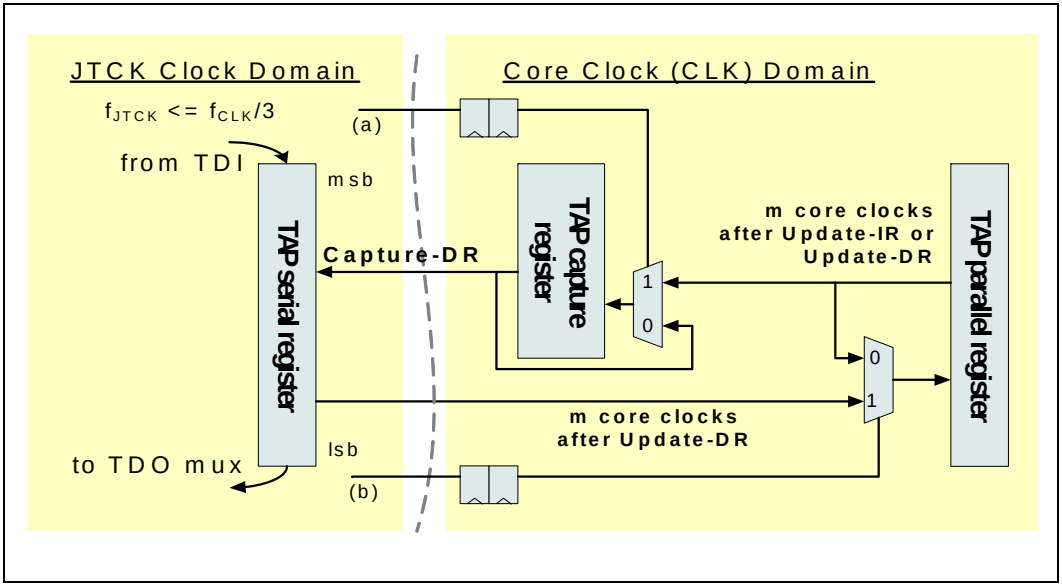


Figure 2–5. Xtensa TAP to Core OCD Module Register Synchronization

On the left side of Figure 2–5, note signals (a) and (b) which pass through synchronizers as they enter the CLK domain. The transitions on signals (a) and (b), and the shifting of the serial register, are controlled by the TAP FSM. Their activity is mutually exclusive because they occur during different FSM states.

The “parallel register” on the right is the functional Debug module register in the CLK domain (examples: DIR*n*, DDR, etc.), and an associated “capture register” which drives the signals across the clock domain crossing to flip/flop inputs in the JTCK domain. The capture register is only loaded from the parallel register when enabled by a the synchronized enable signal (a) from the TAP FSM in the JTAG domain. This occurs in the Update-IR or Update-DR FSM state, depending upon the register being read. Once the capture register has been loaded with the contents of the parallel register, its outputs are stable and may be sampled by logic in the slower JTCK domain without fear of glitch. The serial register is loaded with the values from the capture register during the Capture-DR or Capture-IR FSM state. The paths from the capture register (CLK domain) and the serial register (JTCK domain) are therefore not synchronized. For example, when reading NAR, the NAR seen by the core is copied to its corresponding TAP capture register in the Update-IR state of the NARSEL TAP instruction when it is executed for the first time. Subsequent reads of NAR do not require intervening NARSEL TAP instruction invocations and can be done by traversing only the data branch of the TAP state machine shown in Figure 2–5.

Transferring data from the serial register in the JTCK domain to the parallel register in the CLK domain is similar. The data is stable in the serial register when the TAP FSM asserts signal (b) in the figure, loading the parallel register during the Update-DR FSM State. The actual register update takes place after an unknown number of clock cycles, denoted "m" in the figure, due to the difference between the CLK and JTCK frequencies.

2.3.5 Debug Module Register Access using NAR and NDR

Serial JTAG is converted to memory-mapped space access through the mechanism described in this section.

Following are the parameters of the protocol:

- read and write
- 32-bit data only
- 7-bit address space (up to 128 registers)

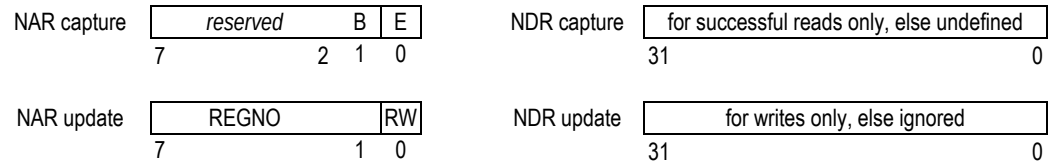
Most registers are accessed in a single cycle. For some registers (currently only TRAXDATA) it is two cycles or more. All accesses have a failure/success (E bit) signaled by the Debug module upon completion of the transaction (via NAR).

Debug module registers are accessible through JTAG with NAR and NDR alternating as the JTAG DR registers. That is, when the TAP IR is set to NARSEL, the TAP DR phase provides access alternately to the NAR and NDR registers, starting with NAR. These “registers” are defined separately on the capture (read) and update (write) sides of a TAP DR shift sequence. Note that when shifting a JTAG TAP register, it is first captured (read), then shifted through the TAP chain, then updated (written), even though the JTAG host writes the data first and reads the data after (shifting swaps read and write data).

The 8-bit NAR, when written (updated), consists of a read/write select bit and 7 address bits, for reading from or writing to one of up to 128 Debug module registers (see Figure 2–2 for detailed address map).

When read (captured), NAR reports status of the last transaction.

When written (updated), the 32-bit NDR writes its 32 bits of data to the register last selected by NAR, if NAR selected a write; otherwise the NDR update is ignored.
When read (captured), NDR reports the results of the register read, if NAR selected a read and the read was successful; otherwise the value returned is undefined.



Bits of NAR are as follows:

RW	Selects register read (RW=0) or write (RW=1).
REGNO	Selects which register to read or write (0 to 127).
E	Error status bit for the previous NAR+NDR read or write transaction. See detailed descriptions of individual debug registers for whether/how this gets set.
B	Busy status bit for the previous NAR+NDR read or write transaction. See detailed descriptions of individual debug registers for whether/how this gets set.

From a programmer's perspective, each NAR and NDR scan can be viewed as individual steps.

For example (E and B bits are a bit simplified here):

- NAR scan:
  - Capture-DR: reports status (E and B bits)
  - Update-DR: updates NAR
    If requests read, initiates read and clears status (E=0 and B=1; B to be cleared and E updated when read considered complete).
    If requests write, no action (NAR indicates that NDR scan will initiate write).
- NDR scan:
  - Capture-DR:
    Reports read data (whether or not valid); may cancel read still in progress.
  - Update-DR:
    If NAR indicates read, no action.
    If NAR indicates write, initiate write with the NDR data just updated, and clears status (E=0 and B=1; B to be cleared and E updated when write considered complete).

2.3.6 JTAG Error (E) Bit

Certain Debug module registers are accessed in a variable number of cycles. Since there is no mechanism to delay JTAG accesses, status bits are needed to manage the fact that accesses to some registers may not complete right away.

Writes to a register may result in the following:

E=0 Write was successful.

E=1 Write was dropped due to timeout (write response not received as of Capture-DR state of the NAR in which we're reporting this bit), or any reason other than "another read or write was pending".

Reads from a register may result in the following:

E=0 Read was successful; data returned was valid.

E=1 Read was dropped due to timeout (read response not received by the time Capture-DR state entered for the NDR shift in which read data is reported), or any reason other than "another read or write was pending".

Currently, the only variable-access time register is TRAXDATA.

2.3.7 TAP Reset

The Xtensa processor core, Debug module, and the TAP controller are independent of one another from the reset perspective. The TAP controller reset has no effect on the core or Debug module operation.

If your logic is driving JTAG, do not drive JTRST with an inverted copy of BReset. Your design must be able to both:

- Assert JTRST without resetting the core (this is a TAP standard requirement).
- Reset the core (assert BReset) without affecting the TAP.

The JTAG specification does not require synchronizing JTRST with JTCK. Nonetheless, de-asserting JTRST must not cause any timing violations, which can occur because TAP controller flip/flops are clocked by JTCK and asynchronously reset by JTRST. The Xtensa processor's TAP controller does not internally synchronize JTRST to JTCK, so any logic driving the JTAG interface must manage the de-assertion of JTRST. If you are designing logic to drive TAP, you can either stop the JTCK clock during JTRST de-assertion, or constrain JTAG interface driving logic so that JTRST de-assertion happens without any timing violation, relative to JTCK edges. Commercial JTAG probes, which connect directly to JTAG header pins, correctly manage this timing relationship.

The core clock CLK must be stable and running freely before the TAP is allowed to come out of TestLogic-Reset state because certain JTCK-domain signals—the TestLogic-Reset state decode in particular—are synchronized to the core clock domain before clearing certain states in the Debug module. A way to ensure this is to hold JTRST asserted for a duration of at least 10 CLK cycles.

The TAP controller must be brought into a known state for correct processor operation when leaving the JTAG interface unconnected. This state can be achieved by ensuring that the TAP state machine is in the TestLogic-Reset state—and remains in the TestLogic-Reset state. The latter will naturally happen if JTMS, JTDI, and JTRST are held at logic '1' as required by the TAP standard (see Section 2.3.2).

2.3.8 Miscellaneous

The TAP standard states that when idling TCK, it should be asserted low to avoid losing state in the test circuitry. It also allows TAP implementations to maintain state when TCK is left high. The TAP controller implements this (maintains its state when JTCK is 1) because it is a static design.

2.3.9 Connecting Multiple Cores on a TAP Scan Chain (MP)

There are various ways in which an Xtensa-based multiple processor (MP) "SoC" or "IC" may be debugged using the JTAG (TAP). The simplest is to provide completely separate TAP ports for each Xtensa core. However, this can quickly become unwieldy when there are many Xtensa processors.

The IEEE 1149.1 TAP standard describes various topologies for connecting together any number of TAP compliant devices to a host device. TAP controllers for multiple Xtensa cores may be connected together in such a topology, along with any other TAP compliant devices. This can reduce the number of pins required to access all the cores, however, there is a trade-off between performance, number of pins, complexity and flexibility for each topology and how it is used.

Two of the simpler topologies are described here. Refer to the TAP standard for more details.

2.3.9.1 Simple Scan Chain

The most straightforward and standard way of interconnecting multiple TAP controllers is along a scan chain. Only the data signals are actually chained. The TDO signal of one device connects to the TDI signal of the next along the chain. The $\overline{\text{TRST}}$, TCK, and TMS signals are bused, all taking their input from the host. This way all TAP controllers follow

the same TAP state transitions synchronously. The host sees five signals, just as it would if connected to a single TAP device: TDI of the first device, TDO of the last device in the chain, and $\overline{\text{TRST}}$, TCK and TMS of all TAP devices. This is illustrated in Figure 2–6.

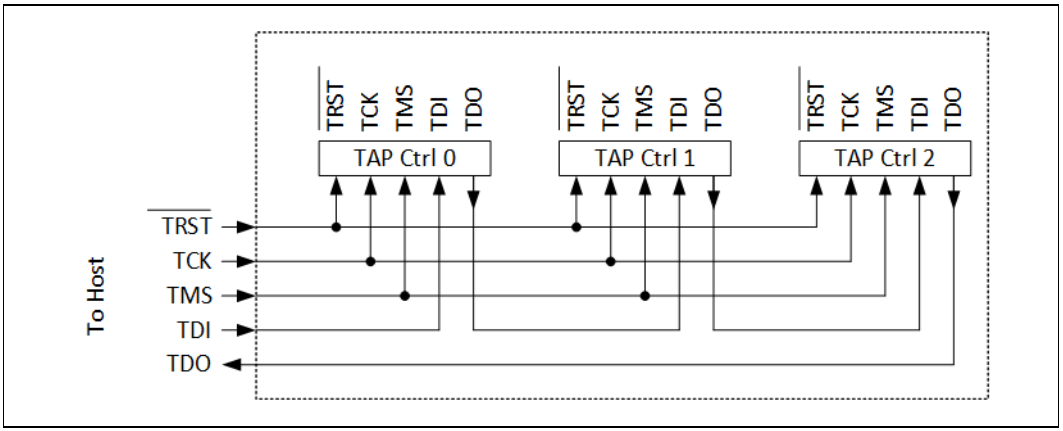


Figure 2–6. Chaining TAP Controllers

Because a simple scan chain looks the same to a host TAP interface as a single TAP controller, it is supported by most existing off-the-shelf host TAP interfaces.

The TAP sequences differ for a chain of TAP devices compared to a single TAP device, at least in the shifting states. All devices' selected serial shift registers are effectively connected end-to-end. So the host must shift one very large aggregated shift register, composed of all the shift registers selected by each TAP device along the chain. This implies that the host must know the relative position of each TAP device along the chain, and some minimum characteristics about each device (at least their instruction register length) so that it knows what to shift.

When in the Shift-IR TAP state (shifting the instruction register), the host must shift an aggregate register composed of an instruction for each TAP device. Since each TAP device must have a fixed length instruction register, the number of bits to shift the aggregate instruction registers is fixed for a given set of TAP devices along a chain. However when in the Shift-DR state (shifting the selected data registers), the length of and function of each TAP device's data register is a function of what instruction they were each given. So for performance and practical reasons, a host will typically set the instruction register of those TAP devices it does not know much about, or is not currently accessing, to BYPASS (an all 1's instruction encoding) which selects a minimal length (single-bit) bypass data register.

Performance Impact

The performance impact of this topology can be costly if there are many devices on the chain. All TAP register transactions are lengthened in proportion to the number of devices on the chain.

This need not be too restrictive. For example, performance would likely be most affected if the Xtensa OCD TAP were chained with all devices on a board to enable more effective board-level (or even chassis-level) testing, because that could lengthen the chain considerably. On a given board, this could be alleviated by placing jumpers (or the equivalent) that allow the processor's TAP to be cut out of the chain and be made available by itself when debugging the processor, or be left in the chain for board/chassis-level testing. Certain board standards, such as the Windows® CE™ Standard Development Board (SDB) Requirements version 5.6, mandate this approach. The next most likely performance issue would be where multiple Xtensa processors on a chip must be accessible via TAPs through a limited number of available pins. Other (non-chaining) alternatives described in this section may help alleviate this issue as well.

Synchronous Start and Stop

In the case of Xtensa, this topology does have the advantage that multiple cores can relatively easily be started or interrupted synchronously, that is, by sending the appropriate instruction to the desired set of cores. This works because the instructions take effect in the Update-IR state for all cores at the same time (after all instructions have been shifted to all the TAP devices in the chain).

2.3.9.2 Parallel TAP Controllers

Another possible topology would be to bus the $\overline{\text{TRST}}$, TCK, and TMS signals as above, but provide separate TDI and TDO signals for each Xtensa core. This way, one retains the performance available with complete separate TAP ports if only one core is debugged at a time (but not if multiple cores are to be debugged simultaneously and independently), with a lesser pin count. However, standard host TAP access interfaces (such as the Macraigor Wiggler) do not directly support such a configuration, so extra work is usually needed. This might be a completely custom host TAP interface, or some method of selecting the desired TDI/TDO pair to a standard host TAP interface. In either case, if a given core's (or set of cores) TAP controller(s) is accessed, the other TAP controllers should usually have their TDI signals left high, causing them to receive the BYPASS instruction.

One interesting advantage of this scheme for multiple Xtensa cores on a single chip is $\overline{\text{TRST}}$, TCK, and TMS signals are bused for all cores and brought out as three pins from the chip, and TDI and TDO signals are brought out individually for each core (see Figure 2–7). In that scenario, one has the option of chaining all cores (or a subset there-

of) in a normal manner, external to the chip; or of avoiding chaining to access a selected individual core with higher performance (assuming unused TDI inputs are pulled high); or of implementing a custom host TAP interface to access multiple cores in parallel.

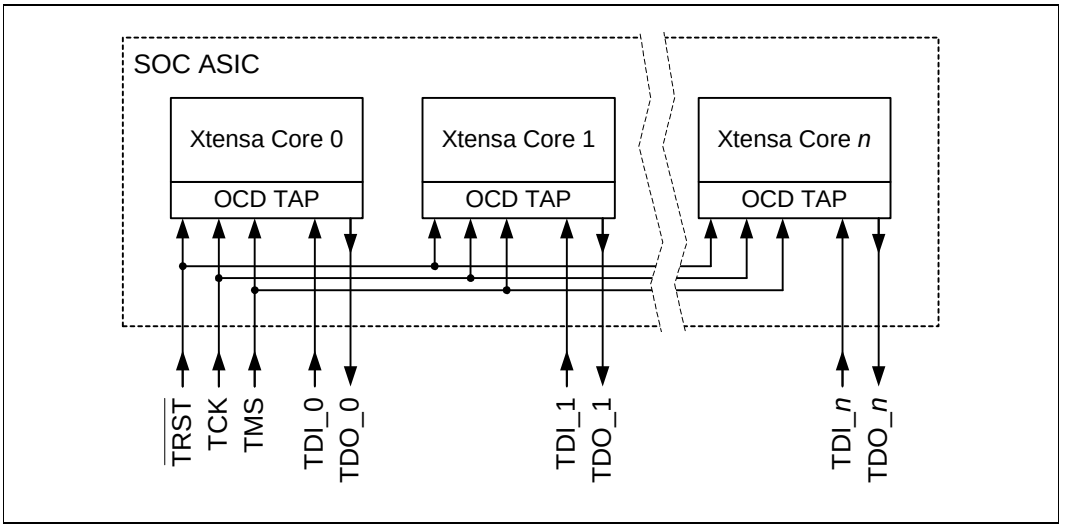


Figure 2–7. Example Reduced SoC Pin Out for Multiple Xtensa Processors

2.3.10 OnCE-Style Connector

Target systems that provide JTAG and TAP interface of Xtensa processor(s) often do so using a standard connector, so as to provide a greater choice of JTAG probe vendors. Currently, the most commonly used connector for Xtensa based development boards is the 14-pin OnCE-style connector, depicted in Figure 2–8.

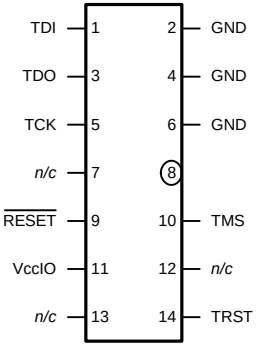


Figure 2–8. OnCE Style Connector Pin-Out

The OnCE[1] style connector is a standard 14-pin male header with 0.1" (2.54 mm) pitch. It contains a superset of the five-signal JTAG TAP interface. The $\overline{\text{RESET}}$ signal on pin 9 allows the JTAG probe to reset the system, that is, the target processor(s) and possibly other target devices. The VccIO signal on pin 11 provides power to the probe from the target (a probe may or may not use this). Pins 7, 12, and 13 are not connected. The connector can be keyed using pin 8 (see below). As shown in Figure 2–8, the remaining five pins are the usual five-signal JTAG interface described elsewhere in this document.

Two methods can be used to ensure the connector is plugged in the correct orientation. The suggested method is to use a shrouded header connector on the target (Figure 2–9). An alternative is to key on pin 8, leaving that pin absent from the male (target) header, and filled on the female (probe) connector. Unfortunately, male headers are generally built with all pins, so the key pin tends to be present unless manually removed. And probes normally ship without the key pin filled, so they can be used with targets that have not removed the header key pin. So a shrouded connector is preferable.

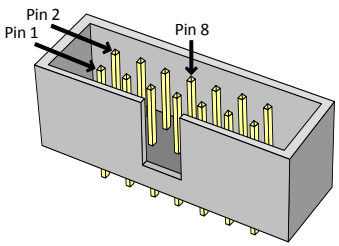


Figure 2–9. Shrouded 14-Pin Header Connector

2.3.11 Connecting the Scan Chain

Using a JTAG interface to the TAP interface allows designers to chain multiple TAP interfaces together. Users may chain together any combination of Xtensa processors and other TAP controllers. The exact wiring for connecting TAP controllers along a scan chain is discussed in more detail in Section 2.3.9. Note that Xtensa OCD Daemon only supports the simple scan chain topology.

Targets using the OnCE-style connector, such as the XT-ML605 and XT-KC705 boards, require wiring additional signals when chained together. Figure 2–10 shows an example schematic for connecting two target boards in a scan chain when using that connector. This can easily be extended to chain more boards:

- The connector's `VccIO` signal must be connected to only one device. Otherwise, tying multiple V_{CC} connections together would cause power supply contention.

1. The connector is referred to as OnCE for historical reasons only: various probe vendors also refer to this connector layout as "OnCE". The JTAG signals supported by Xtensa processors have nothing to do with the OnCE standard. The connector layout just happens to be one used elsewhere for both JTAG and OnCE. For Xtensa processors, only the JTAG and system reset signals are used.

- The $\overline{\text{RESET}}$ lines are shared. Therefore, resetting any board manually will simultaneously reset all the others.

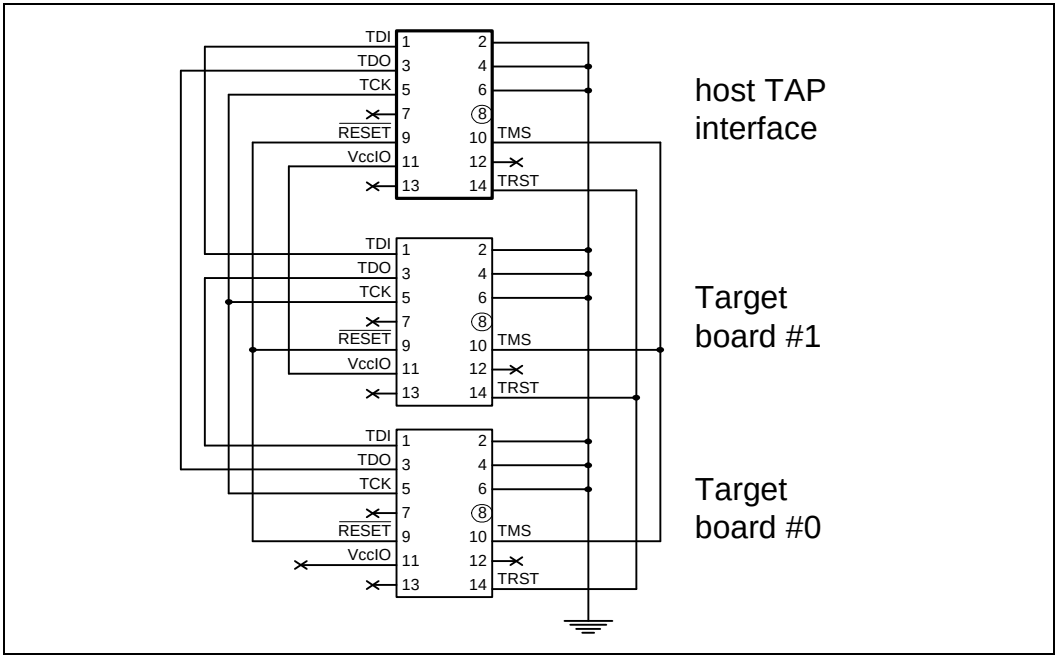


Figure 2–10. Sample MP-OCD Connection for Target Boards Using OnCE Connectors

2.4 APB Interface

The Access Port and Debug modules implement an APB interface. This is a slave interface as specified by the AMBA 3 APB protocol [*AMBA 3 APB Protocol Specification v1.0 Issue B 2004/8/17*]. The slave interface is used to access debug functionality i.e. read and write the OCD, TRAX, and Performance Monitor Module registers.

2.4.1 APB Interface Signals

The signals of the APB interface are shown in Table 2–7.

Note: With Xtensa NX processors, the clocking of the APB interface can be configured to be synchronous to the Xtensa NX core clock.

Table 2–7. APB Interface Signals

Name	Width	Direction	Comments
PBCLK	1	Input	The APB clock. Asynchronous to core clock. Not present in synchronous APB configurations (Xtensa NX option only).
PBCLKEN	1	Input	For asynchronous APB configurations, must be asserted for the duration of APB transactions. Used for power savings when deasserted. Functions as a strobe in synchronous APB configurations (Xtensa NX option only) to allow the APB interface to run slower (at most 16 times slower) than the core clock. Refer to section 13.2 of the *Xtensa NX Microprocessor Data Book*.
PRESETn	1	Input	Resets flops on PBCLK.
PADDR	12	Input	PADDR[13:2] for word-aligned access to 4x4KB, and PADDR[31] to signal external (high) vs. internal (low).
PSEL	1	Input	Select signal.
PENABLE	1	Input	Enable signal (follows PSEL).
PWRITE	1	Input	Distinguishes write vs. read transfer.
PWDATA	32	Input	Write data.
PREADY	1	Output	Allows the slave to add wait states to a transfer.
PRDATA	32	Output	Read data.
PSLVERR	1	Output	Indicates a transfer failure (Debug module reg. access E bit response).

Normally, APB uses the term “PCLK” to refer to the bus clock. However, because this conflicts with PCLK (for Processor Clock) already in use as a virtual reference clock, the Xtensa LX processor uses “PBCLK”, and “PBCLKEN” instead of PCLKEN.

2.4.2 APB Access to Debug Module Registers

Not all Debug module registers are always available to accept APB writes or reads. In that case, the Debug module can insert wait states to the transfer (using PREADY signal) and extend an APB transaction indefinitely.

Some Debug module registers can choose to ignore the transaction if they cannot accept it at the moment. In that case, the APB transfer doesn’t provide any failure response. Debug module registers that can cause the APB transfer failure have their own way of signaling the failure.

2.4.3 APB Clock Domain

For asynchronous APB configurations, APB clock (PBCLK) is assumed to be asynchronous to the Debug module clock (CLK) and APB inputs are synchronized before use within the Debug module. Similarly, APB outputs are synchronized to PBCLK before output. The target registers read/written reside within the OCD, TRAX and Performance Monitor Module functions, and their access mechanisms or FSMs run off CLK.

Due to this synchronization, APB transfers to the Debug module will require more cycles than if a synchronous PBCLK to CLK domain crossing were the implementation.

Even though the APB implementation assumes an asynchronous crossing between the PBCLK and CLK domains, the Xtensa user is free to drive APB with a PBCLK that is synchronous to CLK. However, in such cases the system logic must meet PBCLK static timing requirements.

2.4.4 APB Reset

Resets are configurable as synchronous or asynchronous for Xtensa as a whole. In either case, PRESETn resets only PBCLK-domain flops and logic in the Debug module. A standard Cadence reset synchronizer (synchronous or asynchronous) is used within the Debug module for PRESETn.

When PRESETn is asserted during an APB transaction, the transaction is aborted as far as the APB is concerned. However, the Debug module registers might already have been modified, especially if the abort happens late in the transaction. Due to the asynchronous nature of the CLK-domain to PBCLK-domain crossing, it is impossible for the Debug hardware to time the assertion of PREADY to the APB master with the update of the state. For this reason, when a master aborts a transaction by the assertion of PRESETn, it must check whether the transaction has updated any registers in the Debug module before proceeding to the next transaction. This applies to reads as well as writes because some registers of Debug module can have read side-effects.

2.5 ERI Interface

ERI access allows the Xtensa processor to manipulate its own debug functionality. For example, user applications can measure their performance statistics using the Performance Monitor Module; or debugging an application running on the core can decide when to start or stop tracing; or some registers might be used to communicate between a program running on the core and the software running on a host.

Access to the Debug module and Access Port consists of the use of the RER and WER instructions, for reads and writes, respectively.

With the exception of reading the TRAXDATA register, the ERI provides guaranteed access time to the registers, which is valuable for applications such as a performance interrupt handler running on the core.

ERI reads are speculative, and will be repeated if the RER is replayed for some reason. This may have effect on registers, and the corresponding logic, which implement side-effects on read. Debug registers have no read side-effects.

Debug module registers are able to always accept writes, so the WER will not always be successful. If a write to the register fails, the WRSUC bit of the ERISTAT register is cleared. The bit is set if the ERI write is successful. The bit is intended to be similar to the JTAG transaction success/fail bit E. The ERISTAT register is readable and writable, but writing the register itself through ERI must be done cautiously. The result of the write is not a reflection of the success of the transaction, but simply what was written.

In contrast to writes, ERI reads from the registers that have variable-access time (and cannot accept the read at the time of the request) are extended indefinitely, that is until the read is accepted and the read is available through RER.

Note: The Cadence Instruction Set Simulator (ISS), including XTSC and XTMP modeling APIs, does not currently support the external register interface.

2.5.1 External Register Address Space

The external register space has 16 “device” spaces defined in it as shown in Table 2–8. Only these spaces are used; other addresses are reserved. Each of the 16 device spaces includes a User portion and a Supervisor portion.

Table 2–8. ERI Address Spaces

Beginning Address	Ending Address	Use of Space
0x00000000	0x000FFFFF	Reserved
0x00100000	0x0010FFFF	Device 0 space, Supervisor portion; Debug (i.e. registers of Table 2–1) is device 0
0x00110000	0x0011FFFF	Device 1 space, Supervisor portion; iDMA is device 1
0x001*n*0000	0x001*n*FFFF	Reserved for Device *n* space, Supervisor portion, where *n* is 2 to 16
0x00200000	0x0008FFFF	Reserved
0x00900000	0x0090FFFF	Device 0 space, User portion; Debug has no user registers, therefore this space is unused
0x00910000	0x0091FFFF	Device 1 space, User portion; iDMA is device 1
0x009*n*0000	0x009*n*FFFF	Reserved for Device *n* space, User portion, where *n* is 2 to 16
0x00A00000	0xFFFFFFFF	Reserved

Not all devices have any accessible registers in the User portion; this includes the Debug registers.

For more details on WER/RER privilege and ERI space protection, refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

2.5.2 ERI Inaccessible Registers

Some registers, or register fields, are inaccessible to the ERI. For example, most OCD registers will be inaccessible so that the external debugger is guaranteed true control of the core. The mechanism guaranteeing the inaccessibility is dependent on the hardware implementation in OCD, TRAX and the Performance Monitor Module. In most cases, read returns zero and write has no effect. In all cases, the ERI transaction will complete normally; that is, without error or extra wait states.

3. Reset, Power Control, and Clocking

This chapter describes registers and signals related to reset, power, and clocking as they relate to the Debug module and what it controls.

Section 3.1 describes registers used to manage Debug module and core reset. These registers also manage power. Xtensa LX processors may be configured with a Power Shut-Off (PSO) mechanism in which the Debug module, the core, and its memories are in separately controlled power domains. Section 3.2 describes this PSO mechanism.

Section 3.3 describes the clock gating mechanism, and Section 3.4 describes an additional debug signal related to core reset.

Note: PSO is available for Xtensa LX processors only, and is a feature-controlled portion of the product. To gain access to this functionality, contact your Cadence representative.

3.1 Reset and Power Management Registers

Reset and power are managed using a separate set of control and status registers for each interface to the Debug module: JTAG, APB, and ERI. These registers are located in the Access Port, separately from the Debug module per se, to allow them to control reset and power of the Debug module.

These registers, named `PWRCTL` and `PWRSTAT`, are listed in Table 2–2 on page 7, under the *Misc* module (see Module column), in address range 0x3000 to 0x3BFF. Note that unlike other registers in that table, which are each a single register instance accessible through all three interfaces, the `PWRCTL` and `PWRSTAT` registers exist as separate (identically named) instances for each interface (JTAG, APB, and ERI).

No function-specific Debug registers are accessible over these interfaces while the Xtensa Debug module is powered down. See Section 3.2.1 “Debug Module Power Shutoff” for consequences when erroneous software does so.

For JTAG, `PWRCTL` and `PWRSTAT` are not addressed as other registers. Rather than be accessed using the `NAR/NDR` mechanism, each is a separate JTAG TAP data register, as described in Section 2.3.

For APB, access to `PWRCTL` and `PWRSTAT` is not gated by the `PADDR[31]` bit as other registers are: they are equally accessible by both “internal” and “external” accesses.

3.1.1 PWRCTL – Power and Reset Control Register

Table 3–9, Table 3–10, and Table 3–11 describe PWRCTL register bits for each access interface (JTAG, APB, and ERI, respectively).

Table 3–9. PWRCTL Register (via JTAG)

Bit	Access[1]	Field	Reset Value	Reset By	Description
7	R/W	JtagDebugUse	0	DReset[2] pin	(PSO) Set to enable JTAG access to Debug (Tied high in configurations without PSO.)
6	R/W	DebugReset (JTAG)	0	DReset[2] pin	Setting to 1 asserts reset to the Xtensa Debug module
4	R/W	CoreReset (JTAG)	0	BReset[2] pin	Setting to 1 asserts reset to the core
2	R/W	DebugWakeup (JTAG)	0	PCM	(PSO) Request power to Debug module
1	R/W	MemWakeup (JTAG)	0	PCM	(PSO) Request power to Memory domain
0	R/W	CoreWakeup (JTAG)	0	PCM	(PSO) Request power to core

1. Access: RO means read-only, R/W means read-write. See also “Register Table Terms” on page xxii.

2. Also reset by PCM reset, if PSO is configured.

Table 3–10. PWRCTL Register (via APB)

Bit	Access[1]	Field	Reset Value	Reset By	Description
28	R/W	DebugReset (APB)	0	DReset[2] pin	Setting to 1 asserts reset to the Xtensa Debug module
16	R/W	CoreReset (APB)	0	BReset[2] pin	Setting to 1 asserts reset to the core
12	R/W	DebugWakeup (APB)	0	PCM	(PSO) Request power to Debug module
8	R/W	MemWakeup (APB)	0	PCM	(PSO) Request power to Memory domain
0	R/W	CoreWakeup (APB)	0	PCM	(PSO) Request power to core

1. Access: RO means read-only, R/W means read-write. See also “Register Table Terms” on page xxii.

2. Also reset by PCM reset, if PSO is configured.

Table 3–11. PWRCTL Register (via RER and WER Instructions)

Bit	Access[1]	Field	Reset Value	Reset By	Description
28	R/W	DebugReset (ERI)	0	DReset pin[2]	Setting to 1 asserts reset to the Xtensa Debug module!

Table 3–11. PWRCTL Register (via RER and WER Instructions) (continued)

Bit	Access[1]	Field	Reset Value	Reset By	Description
12	R/W	DebugWakeup (ERI)	0	PCM	(PSO) Request power to Debug module
8	R/W	MemWakeup (ERI)	0	PCM	(PSO) Request power to Memory domain
0	R/W	ShutCoreOffOnPWait	0	PCM	(PSO) Core request to shut off upon WAITI (that is, when asserting PWaitMode)

1. Access: RO means read-only, R/W means read-write. See also "Register Table Terms" on page xxii.

2. Also reset by PCM reset, if PSO is configured.

Following are particulars for the register bits:

- **JtagDebugUse**

  This bit only exists in Xtensa cores configured with PSO with multiple power domains. In other configurations, it is read as 1, and writes to it are ignored.

  This bit must be set prior to accessing Debug module registers over JTAG. Otherwise, JTAG accesses to the Debug module result in undefined behavior. Note that this applies only to the registers physically in the Debug module (which coincides with the Debug PSO domain). As explained in Section 2.2 "Detailed Address Map", this means all the registers in Table 2–2 on page 7, except `PWRCTL` and `PWRSTAT`.

  Only a transition from 0 to 1 allows JTAG access to commence. This bit is reset upon TAP reset. It is also effectively reset when the Debug module is reset; which in turn happens through (1) `PWRCTL.DebugReset` (2) `DReset` (3) PCM wakeup reset. We say "effectively" because the reset happens to a version of this bit in the `CLK` domain as opposed to the originating `JTCK` domain. Finally, any write to `PWRCTL` when `JtagDebugUse` is set also clears the bit.

  Therefore, to properly use Debug registers through JTAG, software must ensure that:

  1. TAP is out of reset,
  2. Xtensa Debug module is out of reset,
  3. Other bits of `PWRCTL` are set to their desired values, and finally
  4. `JtagDebugUse` transitions from 0 to 1.

  This bit must continue to be 1 in order for JTAG accesses to Debug module registers to happen correctly. Once again, when it is set, any write to this bit clears it. Software must ensure that does not happen during JTAG access; or if it does, needs to write 1 again so that JTAG accesses continue.

- **DebugReset**

  This is a set of three independent bits for JTAG, APB, and ERI.

  Setting any of these bits asserts reset to the entire Debug module, including OCD, TRAX, and performance counters. (Which also causes `PWRSTAT.DebugWasReset` to be set.)

  All bits must be zero to de-assert the Debug module reset.

  Software must ensure that this bit remains high for at least 10 processor clock cycles in order for reset to happen correctly. This bit is not cleared automatically, thus allowing debugger software to leave the Debug module in a reset state for an indefinite period of time. After software deasserts this bit, before reading other Debug registers, polling on bit 31 of the Debug Status Register (see Table 5-22) should be performed until it returns 1'b1.

  The JTAG version of this bit is reset to 0 upon TAP reset. The APB and ERI versions of this bit are reset to 0 by PCM reset.

- **CoreReset**

  This is a set of two independent bits for JTAG and APB. (The core cannot reset itself with this over ERI.)

  Setting any of these bits asserts reset to the core. (Which also causes `PWRSTAT.CoreWasReset` to be set.)

  All bits must be zero to de-assert core reset.

  Software must ensure that this bit remains high for at least 10 processor clock cycles in order for reset to happen correctly. This bit is not cleared automatically, thus allowing external debugger software to perform debug accesses while the core is held in reset (e.g., to assert an interrupt) or to leave the core in reset state for an indefinite period of time.

  The JTAG version of this bit is reset to 0 upon TAP reset. The APB version is reset to 0 by PCM reset.

- **DebugWakeup**

  This bit only exists in Xtensa LX processors configured with PSO with multiple power domains.

  This is a set of three independent bits for JTAG, APB, and ERI.

  Setting any of these bits requests the debug domain (Debug module) to be powered on. This includes OCD, TRAX, and performance counters. All bits must be zero, and the `PsoExternalDebugWakeup` external signal be de-asserted, to allow the Debug module to power off.

  After setting one of these bits, the requestor must poll `PWRSTAT.DebugDomainOn` and wait for it to become set to know that the Debug module is powered on.

Once this bit is set, the PWRSTAT.DebugDomainOn bit is not allowed to transition from 1 to 0. Thus, if a store to PWRCTL that sets this bit is immediately followed by a read of PWRSTAT that shows DebugDomainOn is set, the Debug module power can be assumed to be on and remain on (at least until PWRCTL is written to clear it, and all other agents release the Debug module power, etc).

Prior to core shut off, the core can set the ERI version of this bit to leave the Debug module powered up. Note that if it does so, after core shut off there is no way for an external agent to shut off the Debug module. The core needs to be woken up, and then a WER issued to clear the ERI version of this bit.

- **MemWakeup**

  This bit only exists in Xtensa LX processors configured with PSO with multiple power domains.

  This is a set of three independent bits for JTAG, APB, and ERI.

  Setting any of these bits requests the memory domain (local memories and caches) to be powered on. All bits must be zero, and the PsoExternalMemWakeup external signal be de-asserted, to allow the memory domain to power off.

  This only has effect while the core is powered off, as the memory domain is always forced to be powered on while the core is powered on. In effect, when emerging from its reset, the core automatically forces the memory domain to be powered on.

  Prior to core shut off, the core can set the ERI version of this bit to leave the memories powered up. Note that if it does so, after core shut off there is no way for an external agent to shut off memories. The core needs to be woken up, and then a WER issued to clear the ERI version of this bit.

- **CoreWakeup**

  This bit only exists in Xtensa LX processors configured with PSO with multiple power domains.

  This is a set of two independent bits for JTAG and APB.

  Setting any of these bits requests the core to be powered on. All bits must be zero, and the PsoExternalProcWakeup external signal be de-asserted, to allow the core to power off.

  After setting one of these bits, the requestor must poll PWRSTAT.CoreDomainOn and wait for it to become set to know that the core is powered on.

  Once this bit is set, the PWRSTAT.CoreDomainOn bit is not allowed to transition from 1 to 0. Thus, if a store to PWRCTL that sets this bit is immediately followed by a read of PWRSTAT that shows CoreDomainOn is set, core power can be assumed to be on and remain on (at least until PWRCTL is written to clear it, and all other agents release core power, etc).

- **ShutCoreOffOnPWait**

  This bit only exists in Xtensa LX processors configured with PSO with multiple power domains.

  The core sets this bit to request powerdown when it next executes the `WAITI` instruction (meaning, that is, after it next enters `PWaitMode`. The bit is preserved during shut-off, such that when the core emerges from reset, it matches the value of the `PWRSTAT.WakeupReset` bit: it is set on a warm start (wakeup after PSO), and clear on a cold start.

3.1.2 PWRSTAT – Power and Reset Status Register

Table 3–12 (for Xtensa NX processors only), Table 3–13 (for Xtensa LX processors only), Table 3–14, and Table 3–15 describe `PWRSTAT` register bits for each of the available interfaces respectively.

Table 3–12. PWRSTAT Register for Xtensa NX Processors (via JTAG)

Bit	Access[1]	Field	Reset Value	Reset By	Description
6	R/clr	DebugWasReset (JTAG)	0	TAP	Debug module was reset since last time this bit was cleared
4	R/clr	CoreWasReset (JTAG)	0	TAP	Core was reset since last time this bit was cleared
3	RO	CoreStillNeeded	n/a		(PSO) Set if any core wakeup signal is set. Indicates some agent still requires power to the core.
2	RO	DebugDomainOn	n/a		(PSO) Set if debug domain is powered on
1	RO	MemDomainOn	n/a		(PSO) Set if memory domain is powered on
0	RO	CoreDomainOn	n/a		(PSO) Set if core is powered on

1. Access: RO means read-only, R/W means read-write, R/clr means readable and clear bits written as 1. See also “Register Table Terms” on page xxii.

Table 3–13. PWRSTAT Register for Xtensa LX Processors (via JTAG)

Bit	Access[1]	Field	Reset Value	Reset By	Description
6	R/clr	DebugWasReset (JTAG)	1	TAP	Debug module was reset since last time this bit was cleared
4	R/clr	CoreWasReset (JTAG)	1	TAP	Core was reset since last time this bit was cleared
3	RO	CoreStillNeeded	n/a		(PSO) Set if any core wakeup signal is set. Indicates some agent still requires power to the core.
2	RO	DebugDomainOn	n/a		(PSO) Set if debug domain is powered on
1	RO	MemDomainOn	n/a		(PSO) Set if memory domain is powered on
0	RO	CoreDomainOn	n/a		(PSO) Set if core is powered on

1. Access: RO means read-only, R/W means read-write, R/clr means readable and clear bits written as 1. See also “Register Table Terms” on page xxii.

Table 3–14. PWRSTAT Register (via APB)

Bit	Access[1]	Field	Reset Value	Reset By	Description
28	R/clr	DebugWasReset (APB)	1	Debug[2]	Debug module was reset since last time this bit was cleared
16	R/clr	CoreWasReset (APB)	1	Core[2]	Core was reset since last time this bit was cleared
12	RO	DebugDomainOn	n/a		(PSO) Set if debug domain is powered on
8	RO	MemDomainOn	n/a		(PSO) Set if memory domain is powered on
4	RO	CoreStillNeeded	n/a		(PSO) Set if any core wakeup signal is set; indicates some agent still requires power to the core
0	RO	CoreDomainOn	n/a		(PSO) Set if core is powered on

1. Access: RO means read-only, R/W means read-write, R/clr means readable and clear bits written as 1. See also “Register Table Terms” on page xxii.

2. Also reset by PCM reset, if PSO is configured.

Table 3–15. PWRSTAT Register (via RER and WER instructions)

Bit	Access[1]	Field	Reset Value	Reset By	Description
28	R/clr	DebugWasReset (ERI)	1	Debug	Debug module was reset since last time this bit was cleared
12	RO	DebugDomainOn	n/a		(PSO) Set if debug domain is powered on
8	RO	MemDomainOn	n/a		(PSO) Set if memory domain is powered on
4	RO	CoreStillNeeded	n/a		(PSO) Set if any core wakeup signal is set; indicates some agent still requires power to the core
2	RO	CachesLostPower	0	PCM	(PSO) Set if memory domain was powered down while the core was shut off
1	RO	WakeupReset	0	PCM	(PSO) Distinguishes core reset caused by PSO wakeup (1) vs. cold start (0)
0	RO	CoreDomainOn	0	PCM	(PSO) Set if core is reported as powered on; although only readable by the core, thus normally always 1, it reads 0 after wake-up before software clears `PWRCTL.ShutProcOffOnPWait`

1. Access: RO means read-only, R/W means read-write, R/clr means readable and clear bits written as 1. See also “Register Table Terms” on page xxii.

Particular notes about the register bits follow:

- **DebugWasReset**

  This is a set of three independent bits for JTAG, APB, and ERI.

  Each is set to 1 to indicate the Debug module was reset (at least once, by whatever mechanism) since the last time the bit was cleared.

  Each bit is cleared by writing a one. However, it is set as soon as the Debug module is reset again, so in particular, writing a one has no effect if Debug reset is being asserted continuously (for example, by the `PWRCTL.DebugReset` bit being set, or by the external `DebugReset` signal).

  This bit is useful for debug software to recover cleanly (re-synchronize to the target) from the Debug module getting reset outside its control.

- **CoreWasReset**

  This is a set of two independent bits for JTAG and APB. (The core is assumed to know it is reset, by execution of its reset vector.)

  Each bit is set to 1 to indicate the core has been reset (at least once, by whatever mechanism) since last time the bit was cleared.

  Each bit is cleared by writing a one. However, it is set as soon as the core is reset again, so in particular, writing a one has no effect if core reset is being asserted continuously (for example, by the `PWRCTL.CoreReset` bit being set, or by the external `BReset` signal).

  This bit is useful for debug software to recover cleanly (re-synchronize to the target) from the core getting reset outside its control.

- **DebugDomainOn**

  Reports whether the Debug module is powered on. The core, or an APB or JTAG master, can safely access registers of the Debug module when both `DebugWakeup` and `DebugDomainOn` are set.

- **MemDomainOn**

  Reports if local memories are powered on. A host can safely access the memories when both `MemWakeup` and `MemDomainOn` are set.

- **CachesLostPower**

  Allows the core to know whether local memories retained their contents during a core shutdown.

- **WakeupReset**

  Allows the core to know whether — prior to the current reset — it was shut off by the PCM, or whether it is a system cold start.

- **CoreDomainOn**

  Reports if the core is powered on.

  It is safe to assume the core is powered on, and thus accessible, when both `CoreDomainOn` and `CoreWakeup` are set.

3.2 Power Shutoff (For Xtensa LX Processors Only)

The Debug module has a number of interactions with other external modules to ensure that power shutoff happens correctly.

3.2.1 Debug Module Power Shutoff

Debug module shutoff happens when there is no agent requesting that it be up. The power control module (PCM) simply uses its physical shutoff interface to turn power off to the Debug module. However, before this happens, it co-ordinates with the Debug module using the request/acknowledge protocol. The acknowledgment happens when:

- OCD, TRAX, and Performance Monitor Module are all inactive
- The Debug module is not in CoreSight integration mode
- There are no outstanding transactions on APB or ERI directed at Debug registers

Note that JTAG transactions are absent from this list. The TAP needs to remain active during Debug shutoff.

For Debug shutoff to occur, there is an onus on system software to end debug, tracing, and performance monitoring operations and complete ERI or APB transactions. If at the time of shutoff there is an in-flight JTAG read of Debug module registers, it completes with undefined data. An in-flight JTAG write also completes (in the sense that shifting through the TAP FSM happens as expected), but the write to the destination register becomes irrelevant because of the shutoff.

After the system enters into PSO, it should not attempt transactions to the Debug module. If erroneous software does cause transactions to occur:

- JTAG transactions will complete with undefined read and effectless write
- APB transactions will hang
- ERI transactions will complete with read of 0 and effectless write
- Xtensa core access to DDR register will complete with read of 0 and effectless write
- `BreakIn` attempt will hang in the sense that `BreakInAck` will never be asserted
- ATB flush will complete but with no data
- `CrossTriggerIn` attempt will hang in the sense that `CrossTriggerInAck` will never be asserted

When Debug is shut off:

- It does not attempt to issue an OCD instruction to the core
- `BreakOut` or `XOCDMode` is not asserted
- It deasserts `PDebugEnable` which turns off the Traceport

- It does not attempt to read or write the TraceRAM
- It does not issue ATB writes

When the Debug module is in the shutoff state, its response on the various interfaces is altered as described above. Indication that the Debug module is shut off is with DebugDomainOn of the PWRSTAT register as described in Section 3.1.2. The PWRSTAT register is in the Access Port which is in the always-ON domain. The PWRSTAT register is therefore always accessible through JTAG, APB, and ERI.

During PSO, no state is retained in the Debug module. All information is lost, bring up is always from reset.

All resets and clocks are provided by the PCM when waking up. The PCM observes the Debug module requirements on those signals.

3.3 Clocking

The Debug module has three clock domains. The domains are clocked asynchronously of each other, so all clock domain-crossing signals go through synchronizers.

- CLK is the main input clock that goes to both the Xtensa core and Debug Logic.
- JTCK is the standard JTAG clock. Logic in the Debug module used to interface to the TAP controller runs off this clock.
- PBCLK is the APB clock. The APB slave port logic in the Debug module runs off this clock.

3.3.1 Clock Restrictions

CLK, JTCK, and PBCLK are allowed to be asynchronous of each other. However, the following restrictions apply:

- JTCK is required to be no faster than 1/4 the frequency of CLK.
- For Xtensa LX processors: In PSO configurations, PBCLK, CLK, and BCLK must always be toggling. This is because the PCM relies on counting clocks to wake up individual domains. Wake-up will be delayed for as long as there are no edges of the clocks of the given domain.
- Reset signals PcmReset, JTRST, PRESET*n*, BReset, and DReset must be asserted for 10 cycles of the slowest clock among BCLK, PBCLK and CLK. These clocks should be toggling during the entire period.

3.4 OCDHaltOnReset

The Debug module provides an interface to force entry into *Stopped* state (see Section 5.2) when the processor comes out of reset. `OCDHaltOnReset` is a 1-bit input signal to Xttop that is driven by external logic. In most cases, the signal is tied low for normal operation where the Xtensa processor comes out of reset in *Running* state, fetching and executing instructions from memory. Alternatively, the signal can be tied high to force the processor to enable OCD and enter *Stopped* state at processor reset.

This signal may be useful for debugging or to download code using OCD from a separate supervisor processor. Note that `OCDHaltOnReset` is sampled only on Reset and has no effect if it is changed during normal processor operation after Reset is de-asserted.

There are a few rules of timing that must be observed in order for `OCDHaltOnReset` to be correctly sampled. Specifically:

- `DReset` de-assertion must happen at least four `CLK` cycles prior to `BReset` de-assertion. For example, `DReset` is asserted for 10 `CLK` cycles and `BReset` is asserted for 14 CLK cycles.
- `OCDHaltOnReset` must be asserted ten `CLK` cycles prior to the de-assertion of `BReset`.

The reason for these requirements is that `BReset` and `OCDHaltOnReset` can be sampled inside the Debug module only some time after `DReset` is de-asserted.

4. CoreSight Registers

CoreSight registers are common to all functions in the Xtensa Debug module. These registers are listed in Table 4–16. Subsections that follow discuss each register in more detail.

Table 4–16. CoreSight Registers

Address	Register	Access	Rd or rst val	Description
0x3F00	ITCTRL	R/W	0x0	Integration Mode Control Register. [0] = 1 if in integration (i.e. topology-detection) mode, 0 if normal operation
0x3FA0	CLAIMSET	R/set	0xFFFF	Claim Tag Set Register. 16 tags available tags; writing 1 to one of CLAIMSET[15:0] sets the tag, writing 1 to one of CLAIMCLR[15:0] clears the tag.
0x3FA4	CLAIMCLR	R/clr	0x0	Claim Tag Clear Register. 16 tags available tags; writing 1 to one of CLAIMSET[15:0] sets the tag, writing 1 to one of CLAIMCLR[15:0] clears the tag. Reading CLAIMCLR returns the tag values.
0x3FB0	LOCKACCESS	WO	n/a	Writing C5ACCE55h unlocks internal masters, other values lock them
0x3FB4	LOCKSTATUS	RO	0x0	[0] = 0 if read via JTAG (never locked), 1 if read via APB by internal master (lock control mechanism exists) [1] = 0 if read via JTAG, else if read via APB by internal master, 0 if unlocked, 1 if locked (reset/default value is unlocked) [2] = 0 (indicates that this is 32-bit Lock Access Register)
0x3FB8	AUTHSTATUS	RO	0xf0	Authentication Status. Implemented as per CoreSight architecture specification.
0x3FC8	DEVID	RO	see next col	Constant based on endianness of Xtensa - 0x120034e5 if little, 0x121034e5 if big. Same as JTAG IDCODE, different from OCD ID register for OCD component, and TRAX ID register for TRAX/Performance Monitor Module component.
0x3FCC	DEVTYPE	RO	0x15	[7:4] = 1 (SUB Type) [3:0] = 5 (MAJOR Type)
0x3FD0	Peripheral ID4	RO	0x24	[7:4] = 2 (n=2, single 4K*2^n = 16K page for OCD/TRAX/Performance Monitor Module/misc) [3:0] = 4 (number of JEP106 continuation codes, see Section 4.7)
0x3FD4	Peripheral ID5	RO	0x00	Reserved
0x3FD8	Peripheral ID6	RO	0x00	Reserved

Table 4–16. CoreSight Registers (continued)

Address	Register	Access	Rd or rst val	Description
0x3FDC	Peripheral ID7	RO	0x00	Reserved
0x3FE0	Peripheral ID0	RO	0x03	Part Number [7:0] which is 0x03
0x3FE4	Peripheral ID1	RO	0x21	[7:4] = 0x2 (JEP106 identity code [3:0], see Section 4.7) [3:0] = 0x1 (Part Number [11:8], see Section 4.7)
0x3FE8	Peripheral ID2	RO	0x0F	[7:4] = 0x0 (Revision, see Section 4.7) [3:0] = 0xF (JEP106 identity code [7:4], see Section 4.7)
0x3FEC	Peripheral ID3	RO	0x00	[7:4] = 0 (RevAnd; n/a) [3:0] = 0 (Customer Modified – n/a, customer cannot modify Xtensa)
0x3FF0	Component ID0	RO	0x0D	Preamble
0x3FF4	Component ID1	RO	0x90	[7:4] = 9 (Component Class = CoreSight)
0x3FF8	Component ID2	RO	0x05	Preamble
0x3FFC	Component ID3	RO	0xB1	Preamble

4.1 ITCTRL – Integration Mode Control Register

The Debug module supports topology detection. This requires control and status register bits on certain key signals of the Break, ATB, and CTI (Cross Trigger) interfaces. In “integration mode” — as it is known in CoreSight — writes to the control register causes the interface outputs to go to the written value. Similarly, reads of the observation register give the values on the input signals. This scheme allows an external agent to rapidly determine whether two CoreSight components are connected to each other without having to understand or operate the underlying function.

The ITCTRL register is used to enable Integration Mode. Only bit 0 of the register is meaningful, and this is the ITMODE bit which turns on topology detection. In this mode, the topology control register values are muxed into the interface outputs or interface inputs are flopped into the topology observation registers. Table 4–17 and Table 4–18 show the topology control and status bits.

Table 4–17. Topology Detection Control Register Bits

Name	Register	Bit Position	Access	Description
ITBIA	DCR	25	R/W	BreakInAck output control value used in integration mode
ITBO	DCR	24	R/W	BreakOut output control value used in integration mode

Table 4–17. Topology Detection Control Register Bits (continued)

Name	Register	Bit Position	Access	Description
ITATV	TCR	24	R/W	ATVALID output control value used in integration mode
ITCTIA	TCR	23	R/W	CrossTriggerInAck output control value used in integration mode
ITCTO	TCR	22	R/W	CrossTriggerOut output control value used in integration mode

Table 4–18. Topology Detection Observation Register Bits

Name	Register	Bit Position	Access	Description
ITBI	DSR	26	R	BreakIn input observation value used in integration mode
ITBOA	DSR	25	R	BreakOutAck input observation value used in integration mode
ITATR	TSR	24	R	ATREADY input observation value used in integration mode
ITCTI	TSR	23	R	CrossTriggerIn input observation value used in integration mode
ITCTOA	TSR	22	R	CrossTriggerOutAck input observation value used in integration mode

Note that ITATV is special in that it is shared with (i.e. is the same as) ATID[0].

Topology detection register bits are not required for addressable interfaces such as APB or JTAG.

4.2 CLAIMSET, CLAIMCLR – Claim Tag Set and Clear Registers

Often there are a number of debug agents that must cooperate to control the resources that the Debug module makes available. For example, an external debugger and debug code running on another on-chip core might both require control of the TRAX resources. It is important that a debug agent does not reprogram debug resources that another debug agent is using. The Claim Tag Registers provide 16 bits that can be separately set and cleared to indicate if functionality is in use by a debug agent.

Note that the claim tags can be used to coordinate access to registers shared by an external OCD debugger and debugger code running on the Xtensa processor. This is because the core has access to these registers via the ERI.

CLAIMSET and CLAIMCLR are implemented as described in the CoreSight Architecture Specification. A write of 1 to CLAIMSET sets the selected claim tag bits, a read returns the supported tag bits. A write of 1 to CLAIMCLR clears the selected claim tag bits, whereas a read returns current tag bits. Different bits of the registers, that is different tags, may be written at the same time from different interfaces. If the same tag is written at the same time from different interfaces, setting takes precedence over clearing.

4.3 LOCKACCESS, LOCKSTATUS – Lock Registers

The Lock Registers prevent accidental access to the Debug module. A debug monitor must unlock the Debug module before access, and lock it again before exiting. In this way, the software being debugged can be prevented from accessing the OCD, TRAX, or Performance Monitor registers.

LOCKACCESS and LOCKSTATUS are implemented as described in the CoreSight Architecture Specification. Writing the given 32-bit code to LOCKACCESS will unlock internal accesses to Debug module, and any other write will lock internal accesses. External accesses from APB (i.e. PADDR[31] high) are not subject to the locking mechanism, and neither are accesses from JTAG or ERI.

If an internal APB write access should occur to a Debug module register when the LOCKSTATUS indicates that internal accesses are not allowed, the write is ignored by the hardware. APB read accesses have undefined data.

4.4 AUTHSTATUS – Authentication Status

The Debug module supports CoreSight's authentication protocol. The authentication interface signals as shown in Table 4–19 are present as inputs. The Xtensa processor supports the asynchronous version of the CoreSight authentication interface.

Table 4–19. Authentication Interface Signals

Name	Width	From/To	Description
DBGEN	1	System➔Debug	Invasive debug enable
NIDEN	1	System➔Debug	Non-invasive debug enable
SPIDEN	1	System➔Debug	Secure invasive debug enable
SPNIDEN	1	System➔Debug	Secure non-invasive debug enable

Note that in an earlier release of Xtensa LX processors, the core had these pins but they were ignored. In Xtensa LX7 and Xtensa NX1 processors, they must be driven as per Arm's *CoreSight Architecture Specification* for correct access to debug functions.

Present in the Debug module in addition to this interface is the AUTHSTATUS register that reports the implemented security level, and the current status of the enable. The Debug module implements secure debug only, and therefore reports 0x0 for the NSID (bits [1:0]) and NSNID (bits [3:2]) fields of the register. For more details, refer to Arm's *CoreSight Architecture Specification*.

4.4.1 Definitions and Flavors of Access

4.4.1.1 Definitions

What does *debug* mean? Xtensa takes this to refer explicitly to the debug functions OCD, TRAX and PerfMon, and not to other functions that happen to be accessible through the same channels of JTAG, APB, and ERI such as the Power/Reset, iDMA or AXI fault registers.

What does *secure* mean? As per the *CoreSight Architecture Specification*, a non-secure debug operation is any operation where instructions executing on-chip with non-secure privileges or operations external to the system, can cause the same effect. Any other operation is a secure debug operation.

Note: S*ecure* in this context applies only to the use of the Xtensa processor's debug functions. Xtensa has no notion or implementation of a secure zone or state which authenticates all system or software access.

Debug operations that monitor the time taken by a secure routine are not therefore considered secure debug operations, because this can be measured by a combination of off-chip timing and non-secure on-chip event generation. Operations that affect the time taken by a secure routine are considered secure debug operations.

What does *invasive* mean? Again, as per the CoreSight Architecture Specification, any operation that changes the defined behavior of the system is invasive.

This includes any changes to the contents of memory, and insertion of instructions into a processor pipeline. It does not necessarily include effects that change the number of cycles taken to perform an operation, unless the number of cycles is defined architecturally.

4.4.1.2 The Xtensa Implementation

Restating, it is assumed that the authentication interface applies only to debug functions: OCD, TRAX and PerfMon. Other functions are freely accessible regardless of the authentication signal values. This is detailed in Table 4–20.

Table 4–20. Debug Functions Subject to Authentication

Set of Registers	Classification	Subject to authentication?
CoreSight	Non-invasive	No
Misc. - ERISTAT	Non-invasive	No
Misc. - Power/Reset	Invasive	No
OCD	Invasive	Yes
TRAX, PerfMon	Non-invasive	Yes

The authentication function is present only when a CoreSight-compatible debug configuration is selected, which in turn is selected by choosing the APB interface to Xtensa. Table 4–21 describes the flavors of access using the presence of APB as a shorthand for CoreSight-compatible debug.

Table 4–21. Flavors of Access vs. Configuration

Flavor of Access	No APB Configured	APB Configured
Invasive debug e.g. OCD	Always permitted; interface and AUTHSTATUS absent	Permitted if (DBGEN && SPIDEN)
Non-invasive debug e.g. TRAX, PerfMon	Always permitted; interface and AUTHSTATUS absent	Permitted if ((DBGEN || NIDEN) && SPIDEN || SPNIDEN))

4.4.2 Interface Behavior

When invasive debug is not permitted, OCD is disabled. This is equivalent to forcing DCR.EnableOCD low. Writes to the bit from any interface (ERI, APB, or JTAG) will have no effect.

When invasive debug is not permitted, debug interrupts based on BreakIn/Out, TRAX or the host (DCR.DebugInterrupt) are disabled. However, this will not affect real-time vectored debug based on software breaks.

When non-invasive debug is not permitted, TRAX is disabled. This is equivalent to forcing TRAXCTRL.TREN low and TRAXCTRL.TRSTP high. Writes to these bits from any interface (ERI, APB, or JTAG) have no effect.

When non-invasive debug is not permitted, PerfMon is disabled. This is equivalent to forcing PMG.PMEN low. Writes to the bit from any interface (ERI, APB, or JTAG) have no effect.

When invasive debug is not permitted, JTAG, APB, and ERI still work. However, in addition to the fields explicitly mentioned above, all OCD registers and fields will not be writable (that is, write has no effect, and all OCD registers and fields will respond with all zeros when read).

Similarly, when non-invasive debug is not permitted, JTAG, APB and ERI still work. However, in addition to the fields explicitly mentioned above, all TRAX and PerfMon registers and fields will not be writable (that is, write has no effect, and all TRAX and PerfMon registers and fields will respond with all zeros when read).

The authentication signals are allowed to change only when debug functions are not in use. For example, changing the value of DBGEN while halted debug is true (DSR.Stopped) or tracing is ongoing (TRAXSTAT.TRACT high) will result in undefined behavior. It is the responsibility of system software to ensure this.

The authentication signals are flopped inside the Debug module so they will take effect only after two CLK edges. However, as long as the JTAG/ERI/APB write to enable the debug function begins after the value at the Xtensa I/O pin is stable and high — and conversely the value at the Xtensa I/O pin is stable and low after the JTAG/ERI/APB read to confirm that the debug function is disabled — defined behavior is assured.

4.5 DEVID – Device Configuration ID Register

This is the same as the 32-bit value as reported via IDCODE through JTAG (see Section 2.3.4); namely, a constant based on endianness of Xtensa (0x120034e5 if little endian and 0x121034e5 if big endian).

Note that DEVID is available only when the Debug module is powered on. In contrast, IDCODE is available even when the Debug module is powered off, but only to JTAG. In other words, DEVID information is not available to APB and ERI if the Debug module in not powered on.

4.6 DEVTYPE – Device Type Register

The CoreSight device *Major Type* field is 0x3 for a trace source component (which matches TRAX), 0x5 for debug logic (which matches OCD), and 0x6 for performance monitoring. However, TRAX, the Performance Monitor Module, and OCD are combined in a single component, so the distinction cannot be made without creating a duplicate set of CoreSight registers. The Debug module simply reports 0x5 (debug logic) as the device *Major Type*. Device *Sub Type* is 0x1, i.e. *Processor Core*.

4.7 Peripheral ID0 to ID7 Registers

These are all read-only registers with the fixed constants required by CoreSight (see Table 4–16 for the complete map). Some notable fields are as follows:

4.7.1 CoreSight Part Number

The 12-bit part number. Currently, this is chosen to be 0x103. This is similar but not identical to the Part Number-portion (bits [27:12]) of the JTAG IDCODE (see Section 2.3.4).

4.7.1.1 CoreSight Revision

The 4-bit Revision is 0. This may be incremented when Debug module functional changes occur in future releases.

4.7.1.2 JEDEC Manufacturer ID

The JEDEC Manufacturer ID assigned to Xtensa processors is 0x72 in bank 5. When fully encoded, this comes out as a series of 5 bytes: 0x7F 0x7F 0x7F 0x7F 0xF2 (4 continuation codes of 0x7F for bank 5, and 0xF2 is 0x72 with the high bit set).

Note: CoreSight documentation refers to JEDEC Manufacturer ID as JEP106.

4.7.2 Component Identifiers

These are read-only registers with the fixed constants required by CoreSight. The only Xtensa-specific field is ComponentID1—read as 0x9, i.e. the Debug module is a CoreSight component.

5. On-Chip Debug Implementation

Xtensa LX and Xtensa NX processors provide a configuration option for On-Chip Debugging (OCD). This option allows the user to externally gain control of the Xtensa processor, upon request or when a debug exception occurs. The ISA Debug option is a prerequisite for OCD.

This chapter introduces on-chip debugging, discusses configuration requirements, and addresses the software developers of debugger support software for Xtensa OCD.

5.1 On-Chip Debugging

OCD support provides access to and control of the architectural (software-visible) state of the processor through the Access Port's JTAG and APB interfaces (Section 2.3 and Section 2.4 respectively). Through the OCD an external debug agent can:

- Gain control over the processor (stop the processor) either by explicitly generating a debug interrupt or upon any debug exception.
- Read and write any software visible register and/or memory location.
- Resume normal processor execution (let the processor run).
- Communicate with a running processor via the `DDR` register.

Figure 5–11 shows a typical OCD access scenario for a chip containing a single Xtensa core. A debugger host system connects to the Xtensa processor's JTAG TAP through a JTAG debug probe, which is typically an external device.

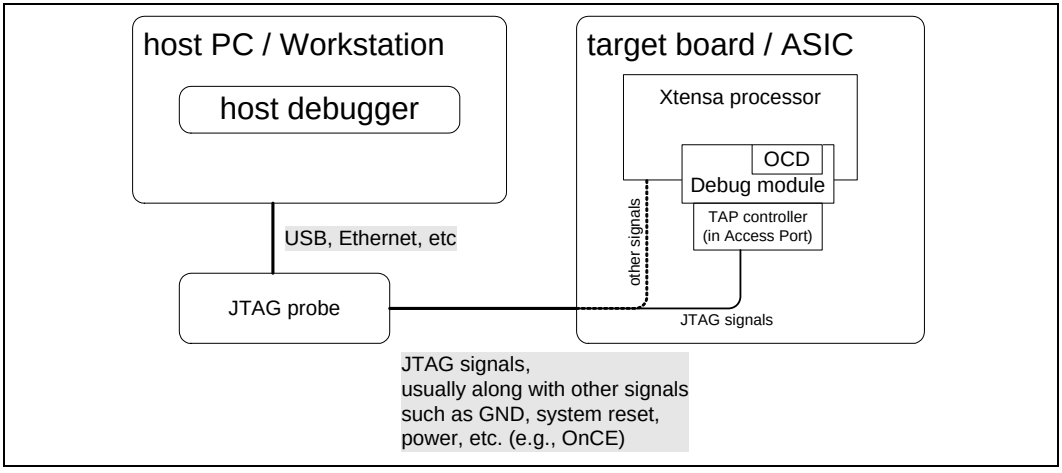


Figure 5–11. Xtensa TAP Overview

5.1.1 Interfaces to OCD Hardware

The following table lists the interfaces from the system in which the Xtensa core resides to the OCD module. The JTAG and APB interfaces are described in Section 2.3 and Section 2.4 of this guide. In this chapter, Section 5.8 describes the Break interfaces, and Section 5.9.3 describes `OCDHaltOnReset`.

Table 5–22. OCD to System Interface

Name	Number of Signals	OCD is	Comments
JTAG	several	slave	This interface comes indirectly through the AccessPort.
APB	several	slave	This interface comes indirectly through the AccessPort.
BreakIn	2	slave	Interrupt from system. See Section 5.8.1 for more details.
BreakOut	2	master	Interface that tells Debug interrupt/exception status. See Section 5.8.2 for more details.
XOCDMode	1	master	Processor is in *Stopped* or *Stepping* state.
OCDHaltOnReset	1	slave	Enable OCD and go to *Stopped* state on processor reset.

5.2 Core Debug States

The Xtensa core may stop on debug events, and step instructions under control of an external debugger, according to whether OCD is enabled or disabled.

Figure 5–12 shows the Xtensa core debug state machine in the context of OCD. The two main states of interest are the *core debug state* (*Running*, *Stopped*, or *Stepping*), and whether OCD is enabled (`DCR.EnableOCD` bit set). Events that trigger changes in state include:

- Debug exceptions in the core (see Section 5.3).
- Debug interrupts from OCD. These include the `BreakIn` signal, TRAX triggers, the `DCR.DebugInterrupt` bit, and the `OCDHaltOnReset` signal.
- OCD request to the core to execute (*step*) an instruction via `DIR0EXEC` or `DDREXEC` registers (see Section 5.5.5 and Section 5.5.6). Note that such an instruction is fetched by the core pipeline from the OCD `DIR` register (see Section 5.5.4).
- Completion of such an OCD stepped instruction.
- Core reset and Debug module reset (see Section 5.9).
- The `OCDHaltOnReset` signal (see Section 5.9.3).

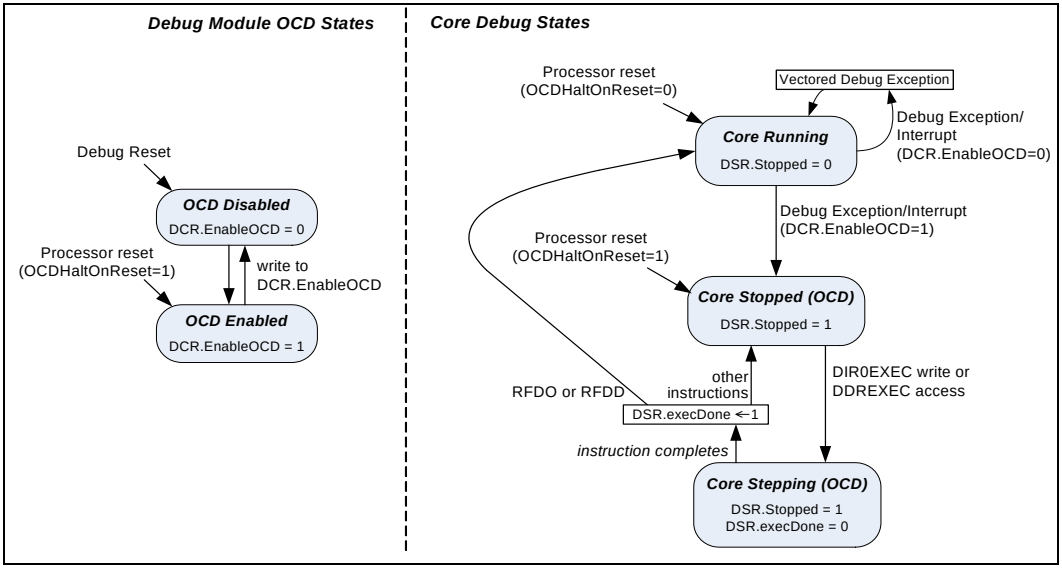


Figure 5–12. Core Debug States

Note how the core behaves differently on debug events (debug exceptions and interrupts; see Section 5.3) according to whether OCD is enabled or disabled.

When OCD is disabled, the core is always in *Running* state, per Figure 5–12: unmasked debug events cause the processor to jump to the debug exception vector and execute the handler. This is called *Vectored Debug* in the terminology of Section 7.1.

When OCD is enabled, unmasked debug events cause the core to break (enter the *Stopped* state) as shown in Figure 5–12. Any committed instructions are completed (the pipeline is flushed), and the core ceases to fetch instructions in the normal manner. At that point, instructions may be directly fed to the core as described in Section 5.2.2, through the *Stepping* state.

OCD is enabled by setting the `DCR` register `EnableOCD` bit to 1 (see Section 5.5.2).

5.2.1 Stopping and Resuming Processor Execution

The processor breaks and enters *Stopped* state upon encountering an unmasked debug event whenever OCD is enabled. While in *Stopped* state, the core no longer fetches or executes instructions. It still responds to OCD executed instructions, which are mostly independent from the core. In general, while in this state an external agent (such as a debugger) has control of the core, using JTAG or APB for example.

Processor execution resumes by executing an RFDO or RFDD processor instruction (these instructions are described in Section 5.7.1 and Section 5.7.2) using the execution mechanism described in Section 5.2.2. This transitions the core state from *Stopped* to *Running* state via the *Stepping* state.

5.2.2 Executing Xtensa Processor Instructions

OCD can fetch and execute almost any Xtensa processor instruction from the DIR register (see Section 5.5.4 for details) rather than from cache or memory through the usual PC-based processor fetch mechanism. By executing sequences of processor instructions through OCD, and communicating data to and from the processor using the DDR special register (Section 5.5.3), a debugger can query and change almost all programmer-visible processor state and memory.

Initiating processor instruction execution moves the core to the *Stepping* state. This state is entered only from the *Stopped* state. While in *Stepping* state, the processor is executing an Xtensa processor instruction that was loaded in DIR. Section 5.5.4 describes how to execute an Xtensa processor instruction using this mechanism.

Only one instruction at a time can be executed in this state — although this may be a FLIX instruction. While the instruction executes, all interrupts are masked (ignored). However, the instruction may take an exception. Once the processor instruction completes normally or encounters an exception, OCD state transitions back to the *Stopped* state; or, in the case of RFDO and RFDD instructions, to the *Running* state (see Figure 5–12). OCD host software normally polls DSR to wait for the instruction to complete (see Section 5.5.1).

5.3 Debug Events (Exceptions and Interrupts) in Xtensa LX Processors

Debug events can be divided into two types: debug exceptions and debug interrupts.

A debug exception is caused by any of the following events associated with the attempted execution of an instruction on the processor:

- a BREAK or BREAK.N instruction is executed
- an instruction address breakpoint (IBREAK) match occurs
- a data address breakpoint (DBREAK) match occurs
- an ICOUNT trap occurs (at PS.INTLEVEL < ICOUNTLEVEL)

A debug interrupt is caused by any one of the following hardware-based events:

- the DCR.DebugInterrupt bit is set to 1; see Section 5.5.2
- the external edge-triggered debug interrupt pin (BreakIn) is asserted; see Section 5.5.1 and Section 5.8.1
- the TRAX processor trigger-out is asserted; see Section 5.5.1
- the OCDHaltOnReset pin is asserted at core reset; see Section 5.9.3

5.3.1 Debug Events when OCD is Disabled

While OCD is disabled (that is, when `DCR.EnableOCD = 0`), any unmasked debug event (either a debug exception or debug interrupt) will cause the processor to jump to the Debug exception vector. The core state is updated as follows:

- DEBUGCAUSE = <debug exception cause>
- EPS[DEBUGLEVEL] = PS
- EPC[DEBUGLEVEL] = PC
- PS.INTLEVEL = DEBUGLEVEL
- PC = DebugExceptionVector

The debug exception vector is also known as the Level-`DEBUGLEVEL` interrupt vector, where `DEBUGLEVEL` equals the “Debug interrupt level” that was chosen in the hardware configuration of that specific Xtensa processor. If OCD is disabled and `PS.INTLEVEL >= DEBUGLEVEL`, then all debug events are masked (ignored).

Note: Due to this masking, it is important not to plant breakpoints in code running with `PS.INTLEVEL >= DEBUGLEVEL`. Such breakpoint instructions are skipped (as a NOP), which corrupts the instruction stream, leading to unpredictable program execution.

5.3.2 Debug Events when OCD is Enabled

While OCD is enabled (that is, when `DCR.EnableOCD = 1`), any unmasked debug events cause the core to break (enter the *Stopped* state). In this case, if `PS.INTLEVEL >= DEBUGLEVEL`, then debug exceptions are masked (ignored), but debug interrupts still cause the core to break and enter *Stopped* state. A debugger can generate a debug interrupt by writing a 1 to the `DCR.DebugInterrupt` bit, thus forcefully breaking the core.

Just after core reset, `PS.INTLEVEL` has a default value of 15. This means that debug exceptions (software breakpoints, hardware instruction and data breakpoints, and `ICOUNT` stepping) will not work until the point in the program that the `PS.INTLEVEL` is lowered below the `DEBUGLEVEL`. If you use the Xtensa OCD Daemon (XOCD), you will still be able to use `ICOUNT` stepping just after reset because XOCD will temporarily re-

duce PS.INTLEVEL below DEBUGLEVEL before it executes the step command. You can also use the debugger to manually lower the value of PS.INTLEVEL so that software and hardware breakpoints are not masked; however, may change the results of your program's execution.

Note: For the same reasons noted in the previous section, it is important not to plant breakpoints in code running with PS.INTLEVEL >= DEBUGLEVEL.

5.3.3 Debug Events when OCD is Stepping the Core

While the core is executing an instruction in the *Stepping* state (that is, when OCD feeds an instruction into the core as described in Section 5.2.2), debug events are always masked (ignored), regardless of the value of PS.INTLEVEL.

Since OCD cannot issue the debug interrupt (CTRL-C) to interrupt an instruction being executed in the *Stepping* state, this might result in a fatal hang of the system if the instruction sent by OCD never completes. In practice, this kind of hang would mainly happen due to an improper external response to a memory access (e.g. a local memory is always Busy, or an AXI slave never responds).

5.4 Debug Events (Exceptions and Interrupts) in Xtensa NX Processors

In Exception Architecture 3 (XEA3) used in Xtensa NX processors, the OCD debugging is no longer associated with a particular interrupt priority level, and it also operates independently from the Monitor Debug mode. There is no basic change to how OCD works in XEA3; however, the way some debug events occur and are handled has changed:

- DEBUGCAUSE register is removed. The information is rolled into EXCCAUSE for Vectored/Monitor Debug mode, and represented in DSR for OCD/halted debug.
- ICOUNT and ICOUNTLEVEL are removed, as these are no longer used for single step operation. Instead, a bit in the OCD DCR register is used for OCD single-stepping, and architectural state is added to the PS register for Monitor Debug mode single-stepping.
- PC and PS are read and written to directly, not via EPC/EPS[DEBUGLEVEL].
- BREAK instructions have a control bit that allows the code to specify if OCD or Monitor Debug mode is to be used. This is used along with the DCR.EnableOCD bit.

Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for details on XEA3.

5.5 OCD Registers

For compatibility with standard memory-mapped device conventions and usage, all registers are 32-bits wide.

Note: The reset value for all OCD registers is 0, with the exception of `OCDID`, `TRAXID`, and `DCR`. The reset values for `OCDID` and `TRAXID` are the constant defined in Section 15.1.1. For `DCR` reset values, refer to Table 5–25.

Table 5–23. OCD Registers

Address (hex)	NAR (hex)	Register	Width	Acc	Description
2000	40	OCDID	32	RO	Same as TRAXID (see Section 15.1.1).
2004	41	–	–	–	(reserved)
2008	42	DCRCLR	bitfield	R/clr	Debug Control Register (write to clear).
200C	43	DCRSET	bitfield	R/set	Debug Control Register (write to set).
2010	44	DSR	bitfield	R/clr	Debug Status Register.
2014	45	DDR	32	R/W	Debug Data Register, used for host to/from target data transfers.
2018	46	DDREXEC	32	R*/W	Alias to DDR; executes the instruction in DIR when accessed.
201C	47	DIR0EXEC	24 or 32	R/W	Alias to DIR0; also executes the instruction when written.
2020	48	DIR0	24 or 32	R/W	Debug Instruction Register; up to 32 first bits of an instruction.
2024 - 203C	49 - 4F	DIR1 - DIR*n*	32	R/W	Debug Instruction Register portion of instructions wider than 32 bits. *n* = (max instruction width / 32) – 1
2040 - 205C	50 - 5F	–	–	–	(reserved for 512-bit FLIX – read as zero, ignore writes)
2060 - 207F	–	–	–	–	(reserved – read as zero, ignore writes)
2100 - 2FFF	–	–	–	–	(reserved – undefined, may alias the above)

5.5.1 DSR – Debug Status Register

The `DSR` register provides OCD status to host debugger.

Not all bits are writable over ERI. See the ERI column (Access heading) in Table 5–24.

Table 5–24. DSR Register Fields

Bit	Field	Access		Description
		APB /JTAG	ERI	
0	ExecDone	R/clr	RO	Instruction in DIR completed execution (w/ or w/o exception)
1	ExecException	R/clr	RO	A previous instruction in DIR completed with an exception
2	ExecBusy	RO	RO	Core is executing DIR (meaningful while Stopped is 1)
3	ExecOverrun	R/clr	RO	DIR execution attempted while previous execute still busy
4	Stopped	RO	RO	Core is under OCD debug control, in *Stopped* or executing DIR
8:5	StopCause (XEA3)	RO	RO	Reason for stopping.
9				(reserved)
10	CoreWroteDDR	R/clr	RO	Core wrote to DDR, i.e. executed WSR.DDR or XSR.DDR
11	CoreReadDDR	R/clr	RO	Core read from DDR, i.e. executed RSR.DDR or XSR.DDR
13:12				(reserved)
14	HostWroteDDR	R/clr	R/clr	Host wrote to DDR (via JTAG or APB) (includes DDREXEC)
15	HostReadDDR	R/clr	R/clr	Host read from DDR (via JTAG or APB) (includes DDREXEC)
16	DebugPendBreak	R/clr	R/clr	Debug interrupt pending due to BreakIn signal
17	DebugPendHost	R/clr	R/clr	Debug interrupt pending due to DCR.DebugInterrupt
18	DebugPendTrax	R/clr	R/clr	Debug interrupt pending due to TRAX PTO
19				(reserved)
20	DebugIntBreak	R/clr	R/clr	Debug interrupt taken due to BreakIn signal
21	DebugIntHost	R/clr	R/clr	Debug interrupt taken due to DCR.DebugInterrupt signal
22	DebugIntTrax	R/clr	R/clr	Debug interrupt taken due to TRAX PTO
23	RunStallToggle	R/clr	R/clr	Set when RunStall input to Xtensa changes polarity.
24	RunStallInputValue	RO	RO	Provides the real-time value of the RunStall input to Xtensa
25	BreakOutAckITI	RO	RO	Break-out topology detection status bit (BreakOutAck signal state)
26	BreakInITI	RO	RO	Break-in topology detection status bit (BreakIn signal state)
30:27				(reserved)
31	1	RO	RO	Always 1. (Read as zero when the Debug module is powered off.)

ExecDone
Indicates that the instruction in DIR completed execution (whether or not with an exception).
Cleared by writing a 1 or by initiating DIR execution; set upon execution completion (after successful DIR execution initiation, whether or not the instruction completed successfully).

ExecException
Indicates that some previous instruction in DIR completed by taking an exception. Set when such an instruction takes an exception; cleared only by writing a 1 to this bit or by entering *Stopped* state i.e. Stopped bit becomes set (in particular, initiating DIR execution does not clear this bit).

Note that the intent is that it be possible for debug software to determine which exception occurred. When an instruction executed via DIR takes an exception, the exception semantics take effect, that is, processor state is updated: PS, EPC (EPC*n* or MEPC or DEPC), EPS*n* if needed, and other registers (EXCCAUSE, EXCVADDR, DEBUGCAUSE, etc). PC value is not changed.
When ExecException is set, any attempt to execute an instruction using DIR, or to write DIR, is ignored (without any effect on ExecBusy, ExecDone, ExecOverrun, or DIR).

ExecBusy
Core is executing DIR (meaningful while Stopped is set, else is undefined and should be zero).
Note: If debugger software wants to be sure its request to execute DIR was accepted, it can clear ExecDone and ExecException before issuing the request; that way, at least one bit is guaranteed to change once the request is accepted (ExecBusy, ExecDone, or ExecException).

ExecOverrun
Set when DIR execution or any DIR write is attempted while a previous one is still being executed, i.e. while ExecBusy is still set (it, and all subsequent instruction execution attempts while this bit was still set, are ignored). Cleared by writing a 1, or when core breaks due to a debug exception while OCD is enabled.

When ExecOverrun is set, any attempt to execute an instruction using DIR, or to write DIR, is ignored (without any effect on ExecBusy, ExecDone, ExecException or DIR).

Stopped
Core is under OCD debug control, stopped or executing DIR (in either *Stopped* or *Stepping* states).

StopCause
(XEA3 only). Reason for stopping (in combination with DebugInt* bits) is under OCD debug control, stopped or executing DIR (in either Stopped or Stepping states).

- 0: None (can occur when DebugInt* bits are set)
- 1: Single-Step Completed
- 2: Hardware Breakpoint Hit (IBREAKn Match)
- 3: (reserved)
- 4: Software Breakpoint Hit (BREAK.1 Instruction)
- 5: Software Breakpoint Hit (BREAK.N Instruction)
- 6: Software Breakpoint Hit (BREAK Instruction)
- 7: (reserved)
- 8: Hardware Watchpoint Hit (DBREAK0 Match)
- 9: Hardware Watchpoint Hit (DBREAK1 Match)
- 10-15: (reserved)

CoreWroteDDR
Core wrote to DDR, i.e. executed WSR.DDR or XSR.DDR. This bit is updated only when the WSR or XSR has successfully retired. To ensure that has happened, software may use the ISYNC instruction.

CoreReadDDR
Core read from DDR, i.e. executed RSR.DDR or XSR.DDR. This bit is updated only when the WSR or XSR has successfully retired. To ensure that has happened, software may use the ISYNC instruction.

HostWroteDDR
Set when DDR written by external host (i.e. via JTAG or APB).

HostReadDDR
Set when DDR read by external host (i.e. via JTAG or APB).

DebugPendBreak
This bit is 1 when BreakIn debug interrupt is pending. It is set when BreakIn interrupt is latched. The bit is cleared either when the debug interrupt is taken (whether or not the OCD is enabled) and indicated by DEBUGCAUSE.DI, or by writing a 1 to this bit.
Note: The core's decision to take the debug interrupt occurs before the automatic clearing of DebugPendBreak and setting of DebugIntBreakIn. If a 1 is written to DebugPendBreak (to clear it) between these two events, DebugIntBreak ends up not getting set even though the debug interrupt is taken. Thus, a debug interrupt can be spurious (no DebugInt*** bits set).

DebugPendHost
Set when a debug interrupt is pending due to writing a 1 to DCR.DebugInterrupt. Reading this bit reports the same value as DCR.DebugInterrupt. Cleared either when the debug interrupt is taken (whether or not DCR.EnableOCD is set) and indicated by DebugInterrupt.DI, or by writing a 1 to this bit.

DebugPendTrax
Set when a debug interrupt is pending due to TRAX processor trigger-out (PTO). Cleared either when the debug interrupt is taken (whether or not OCD is enabled) and indicated by DebugInterrupt.DI or by writing a 1 to this bit.

DebugIntBreak
Set when a debug interrupt is taken due to BreakIn (at which point DebugPendBreak gets cleared). Cleared by writing a 1 to this bit. If DebugPendBreak is cleared just prior to the interrupt being taken (within three cycles) DebugIntBreak will not be set.

DebugIntHost
Set when a debug interrupt is taken due to DCR.DebugInterrupt being set (at which point DCR.DebugInterrupt and DCR.DebugPendHost get cleared). Cleared by writing a 1 to this bit. If DebugPendHost is cleared just prior to the interrupt being taken (within three cycles) DebugIntHost will not be set.

DebugIntTrax
Set when a debug interrupt is taken due to TRAX processor trigger-out (PTO) (at which point DCR.DebugPendTrax gets cleared). Cleared by writing a 1 to this bit. If DebugPendTrax is cleared just prior to the interrupt being taken (within three cycles) DebugIntTrax will not be set.

RunStallToggle
The bit is set on both low to high and high to low transitions of RunStall. Clearable by a write of “1”.

RunStallInputValue
Provides the realtime value of the RunStall input to the Xtensa processor. Since this value could be changing frequently and in a manner asynchronous to the read of the DSR, this bit is used in conjunction with RunStallToggle to attempt to determine a steady-state value of the RunStall input.

BreakOutAckITI
Reports the value of the BreakOutAck signal (see Section 5.8.2). Used for topology detection in the CoreSight environment.

BreakinITI
Reports the value of the BreakIn signal (see Section 5.8.1). Used for topology detection in the CoreSight environment.

5.5.1.1 Debug Pending Bits

The three debug interrupt pending bits (DebugPendBreak, DebugPendHost, DebugPendTrax) allow polling for debug interrupt events, rather than having to take interrupts. They allow a debugger that already has control to detect BreakIn and other such events. The debugger might use this to display debug event information while the core is *Stopped*. For example, late BreakIn may occur with externally managed cross-triggering (e.g. using the Cross Trigger Module (CTM) of CoreSight) or because a core could not process its own BreakIn right away (e.g. was powered off, or had disabled debug interrupts for a long time if not using OCD, etc). DebugPendHost can occur due to other APB masters trying to debug-interrupt this core.

When debugging without OCD (using *Vectored Debug*, for example in a target resident debug monitor), the debug pending bits allow polling for debug interrupts. For example in long operations that keep interrupts disabled thus masking debug interrupts, it allows periodically checking whether a debug interrupt is pending without having to lower the interrupt level.

5.5.1.2 Debug Interrupt Cause Bits

The three debug interrupt cause bits (DebugIntBreakIn, DebugIntHost, DebugIntTrax) report the detailed cause of a debug interrupt. (Indicated with DEBUGCAUSE.DI bit set.)

Although often only one of these bits is set, it is possible for more than one to be set (where multiple interrupt types were pending at the time the debug interrupt was taken). It is also possible for none to be set (i.e. a “spurious” debug interrupt) if the pending bit(s) are cleared just after the core has decided to take the interrupt, and before it clears the pending bits and sets the corresponding debug interrupt cause bits (this is described in the descriptions of each debug interrupt pending bit).

The debug interrupt cause bits help the host software decide how to deal with an interrupt during MP debug.

Note: The core’s decision to take a pending debug interrupt (e.g. DebugPendTrax) occurs before the automatic clearing of the pending interrupt bit and setting the corresponding debug interrupt cause bit (e.g. DebugIntTrax). If a 1 is written to the pending debug interrupt (to clear it) between these two events, the corresponding debug interrupt cause bit ends up not getting set even though the debug interrupt is taken. Thus, a debug interrupt can be spurious (no debug interrupt cause bits is being set to indicate that an interrupt actually happened).

5.5.1.3 Polling for Long Latency Loads

Due to the time it takes to read DSR after an instruction is executed, the Xtensa processor instruction will generally always have completed (successfully or by taking an exception) by the time DSR has been sampled for the first time. However, a very long latency instruction may require polling DSR to wait until completion. There are at least three cases where this may occur: long latency loads, blocking queue instructions, and the WAITI instruction.

DSR Polling for Long Latency Loads

The Xtensa bus (AXI) allows an external device to delay response to a read access for an arbitrarily long time. Usually a system would be designed such that no read access takes longer than a certain number of cycles to complete, whether by ensuring all devices respond within a certain time or by implementing a bus timeout mechanism. In this case, the maximum number of times to poll DSR for a load instruction can be computed. Otherwise, polling could be indefinite. OCD debugger software might choose to implement a software timeout on such polling, and report an error (or warning) when exceeded. Recovering from such a situation (specifically, aborting the instruction) may require resetting the core.

Blocking Queue Instructions

Similarly to loads, queue instructions may stall indefinitely, waiting for activity on the relevant queue. However, unlike load instructions queue instructions may potentially stall for very long periods of time. Although they are interruptible when fetched and executed normally, there is no mechanism to interrupt them when executed in the *Stepping* state. Thus, it is generally not appropriate to execute them that way.

DSR Polling for the WAITI Xtensa Instruction

The WAITI instruction should not be executed in the *Stepping* state. It sets PS.INTLEVEL as requested, and waits for a valid and enabled interrupt to occur before continuing. A WAITI instruction executed in the *Stepping* state will never complete since no interrupts can be taken in this state. A reset of the core will be required.

5.5.2 DCR – Debug Control Register (DCRCLR and DCRSET)

The DCR is used by OCD host debugger software to control and configure the OCD.

It is also meant for functionality accessible by vectored debuggers (such as XMON), via RER and WER (ERI). Some bits are only meaningfully accessible by a debugger external to the core; for this reason and for cleaner external debug control of a core, only some of the bits are writable by the core using WER (over ERI).

DCR is accessible (and equally readable) at two addresses, DCRCLR and DCRSET, for respectively clearing and setting selected bits of the register. Bits written to DCRCLR as 1 are cleared (made 0), and bits written to DCRSET as 1 are set (made 1). Bits written to either address as 0 are left unchanged. In other words, writing zero to either register address has no effect. This scheme allows accessing individual bits without using a software read-modify-write sequence, thus avoiding race conditions when multiple masters access different DCR bits at the same time.

Refer to Table 5–25 for Xtensa NX processors, and Table 5–26 for Xtensa LX processors.

Table 5–25. DCR Register Fields for Xtensa NX Processors

Bit	Field	Access		Reset	Description
		JTAG/APB	ERI		
0	EnableOCD	R/W	RO	0	Set to activate the OCD.
1	DebugInterrupt	R/set	RO	0	Set to break the core (same as DSR.DebugPendHost)
2	InterruptAllConds	R/W	RO	0	Set to allow debug interrupts to supersede all conditions.
3	StepRequest (XEA3)	R/W	RO	0	Request Single Step.
15:4					(reserved)
16	BreakInEn	R/W	R/W	0	Enable BreakIn
17	BreakOutEn	R/W	R/W	0	Enable BreakOut
19:18					(reserved)
20	DebugSwActive	R/W	R/W	0	A software-set flag that indicates user-controlled debug mode.
21	RunStallInEn	R/W	R/W	0	Enable the RunStall input.
22	DebugModeOutEn	R/W	R/W	1	Enable the XOCDMode output.
23					(reserved)
24	BreakOutITO	R/W	R/W		
25	BreakInAckITO	R/W	R/W	0	BreakInAck topology detection control bit
31:26					(reserved)

Table 5–26. DCR Register Fields for Xtensa LX Processors

Bit	Field	Access		Reset	Description
		JTAG/APB	ERI		
0	EnableOCD	R/W	RO	0	Set to activate the OCD.
1	DebugInterrupt	R/set	RO	0	Set to break the core (same as DSR.DebugPendHost)
2	InterruptAllConds	R/W	RO	0	Set to allow debug interrupts to supersede all conditions
15:3					(reserved)
16	BreakInEn	R/W	R/W	0	Enable BreakIn
17	BreakOutEn	R/W	R/W	0	Enable BreakOut
19:18					(reserved)
20	DebugSwActive	R/W	R/W	0	A software-set flag that indicates user-controlled debug mode
21	RunStallInEn	R/W	R/W	1	Enable the RunStall input
22	DebugModeOutEn	R/W	R/W	1	Enable the XOCDMode output
23					(reserved)
24	BreakOutITO	R/W	R/W		
25	BreakInAckITO	R/W	R/W	0	BreakInAck topology detection control bit
31:26					(reserved)

EnableOCD
OCD functionality enabled (Debug exceptions break the core).
Note: If Debug module power is off, EnableOCD is forcefully zero, so that debug exceptions are always vectored (handled in the debug exception vector).

DebugInterrupt
This bit indicates whether a debug interrupt to the core is pending. It is set by writing a 1. Writing a zero has no effect. It is automatically cleared when the debug interrupt is taken (whether by breaking the core if EnableOCD is set, or taking a debug exception if not). It can also be cleared by writing a 1 to the DSR.DebugPendHost bit (which mirrors this bit).

InterruptAllConds
When — for the purposes of obtaining debug control over the processor — a debug interrupt is asserted e.g. by asserting BreakIn, it is accorded the highest priority. In some cases however (e.g., load from a read-side-effect I/O device), it may have to wait for the pipeline to complete the in-flight instruction so as to maintain functional correctness of the system. If this bit is set, the pipeline is forced to kill the instruction and take the debug interrupt, even if it means that the state of the system becomes corrupted.

StepRequest
For Xtensa NX processors. This bit will request single-step of the next instruction, interrupt, or exception. The bit is readable/writable by software running on the Xtensa processor through the ERI (external register interface), or by an external agent through JTAG or APB.

BreakInEn
This bit enables the `BreakIn` input of the Xtensa processor. The bit is readable/writable by software running on the Xtensa processor through the ERI (external register interface), or by an external agent through JTAG or APB.

BreakOutEn
This bit enables the `Breakout` output of the Xtensa processor. The bit is readable/writable by software running on the Xtensa processor through the ERI (external register interface), or by an external agent through JTAG or APB.

DebugSwActive
This bit is set by software to indicate that the processor is in a user-controlled debug mode. By setting the debug mode in this manner, debug operations (for example, from a debugger running on an external host) would be permitted even if the core has RunStall asserted. The bit is readable/writable by software running on the Xtensa processor through the ERI (external register interface) or by an external agent through JTAG or APB. Typically, the external agent would be a debugger running on a host. For the RunStall signal to cause a pipeline stall, this bit must be cleared. This bit is cleared when either the Debug module or the core are reset. Setting this bit also causes the Xtensa output XOCDMode to be asserted.

RunStallInEn
This bit enables the RunStall input to Xtensa. If this bit is cleared, the pipeline will not stall even if RunStall is asserted to the Xtensa processor. This bit would be cleared in scenarios where the user requires the core to make broad execution progress. For example, core A is in debug mode, and all other cores are stalled because the XOCDMode of core A is connected to all other cores' RunStall inputs. Now if you connect with a debugger to core B and not only want to do single stepping, but want to step over functions, i.e., the debugger sets a breakpoint in some removed (i.e. remote) section of the code and does a "continue" Here you would want to ignore the RunStall input signal for an extended period of time. This bit is set when either the Debug module or the core are reset.

DebugModeOutEn
This bit enables the XOCDMode output of the Xtensa processor. If this bit is cleared, XOCDMode is not asserted to the external system even when the core is in debug mode (the DebugMode output will remain asserted in this situation). When set, XOCDMode output signals whether the processor is in debug mode or not. This bit is set when the Debug module is reset so as not to hinder debug-from-reset scenarios.

BreakOutITO
When the CoreSight ITCTRL register is set, this bit controls the state of the BreakOut signal (Section 5.8.2). Otherwise, this bit has no effect. It is used for break in/out network topology detection in the CoreSight environment.

BreakInAckITO
When the CoreSight ITCTRL register is set, this bit controls the state of the BreakInAck signal (Section 5.8.1). Otherwise, this bit has no effect. It is used for break in/out network topology detection in the CoreSight environment.

5.5.3 DDR – Debug Data Register

The Debug Data Register (DDR) is a 32-bit data register that provides a means of communication between the host software and the core, or between the host software and the processor instructions it executes via DIR in the *Stepping* state. A value written into DDR using the WSR instruction can be subsequently read by host debug software over JTAG or APB. Conversely, host software can write DDR using JTAG or APB, which can be subsequently read by the core using the RSR Xtensa instruction.

DDR is not accessible over ERI. Instead, the core accesses DDR as a special register.

When the processor is stopped, DDR is typically used to access and/or change program visible state or memory locations (see Section 7.3.5).

Note: Because the Debug module resides in a different power, reset, and clock-gating domain from the core, the following caveat applies when accessing the DDR using special register access instructions (XSR, WSR or RSR): If the Debug module is shut off, or held in reset or in a clock-gated state — which would happen if no debug functions were active, thereby causing the Debug module to quiesce — accesses by the core are ignored. In other words, DDR reads by the core return spurious data and writes are ignored. The processor does not hang on DDR accesses.

5.5.3.1 Other Effects

Writing DDR via APB or JTAG also sets DSR.HostWroteDDR. This allows target (core) software to reliably detect whether DDR was updated when communicating with the host independent of the host (e.g. for DDR based vectored debug; see Section 7.1).

5.5.4 DIR – Debug Instruction Register

The Debug Instruction Register (DIR) contains an Xtensa processor instruction to execute under OCD control, rather than through the normal processor fetch mechanism.

5.5.4.1 Requesting DIR Execution

Typically, execution is initiated in one of three ways:

- By writing an instruction to DIR register followed by writing data to DDREXEC register (Section 5.5.6). This automatically executes the instruction stored in the DIR register.
- By writing an instruction to DIR0EXEC register (Section 5.5.5). This automatically executes the written instruction.
- Reading from DDREXEC. This requests execution of the instruction already contained in DIR. (DDR is read before the instruction can change it).

Any such request is ignored if the DSR.ExecException or DSR.ExecOverrun bit is set. It is also ignored if the processor is not stopped, i.e. if the DSR.Stopped bit is clear.

If the processor is still busy executing an instruction from DIR (and thus the DSR.ExecBusy bit is set), the DSR.ExecOverrun bit transitions to being set and the request is otherwise ignored (except if reading from DDREXEC in which case the host is expected to first poll the DSR.ExecBusy and ensure no instruction is executing), and the request is otherwise ignored (the instruction in progress is not affected). In other words, only one instruction at a time can be executed via DIR, and any attempt to execute one while another is still executing, is considered an *overrun* (after which no further instructions can execute until the overrun condition is cleared).

Note: Reading DDREXEC always returns the current value of DDR even if the instruction execution request is ignored.

Otherwise, i.e. if the request made it through the above checks, the processor:

- Clears DSR.ExecDone
- Sets DSR.ExecBusy
- Initiates execution of the instruction in DIR.

5.5.4.2 DIR Execution

When the instruction is executed, the core enters the *Stepping* state (DSR.ExecBusy is set) and the processor pipeline gets an instruction from DIR rather than normal fetch paths such as cache or memory.

Most instructions complete in a small number of processor cycles. Some instructions wait for events external to the processor, and thus may take arbitrarily long to complete: for example, push to a full TIE queue, pop from an empty TIE queue, load or store from/to a local memory continually reporting busy or from/to an AXI slave never acknowledging the request, etc.

All interrupts are masked during instruction execution, *including NMI*, with one possible exception: setting both DCR.DebugInterrupt and DCR.EnableOCD interrupts the instruction being executed via DIR, regardless of CINTLEVEL.

5.5.4.3 DIR Execution Completion

The instruction may complete normally, or with an exception. Once it completes, regardless of the cause, the processor:

- Sets DSR.ExecDone
- Clears DSR.ExecBusy
- Sets DSR.ExecException if the instruction completed due to an exception or interrupt.

Note: If the instruction was RFDO or RFDD, the processor also clears DSR.Stopped bit and resumes normal execution. This case never involves setting DSR.ExecException.

5.5.4.4 Exceptions and Interrupts

If an instruction completes with an exception or interrupt, as indicated by DSR.ExecException being set, the instruction generally has not executed. However, exception or interrupt semantics are performed. For example, the relevant EPC and other registers are updated. This however does not include updating the PC register.

Note: Executing instructions via DIR does not update PC as in the normal fetch (non-debug-stopped) case. In other words, PC is not updated to point to the next PC (including, no zero-overhead loopback processing etc). However, PC does get updated to point to the vector on exceptions and interrupts, and on branching instructions that explicitly set PC (e.g. branches, jumps, calls, returns).

5.5.4.5 DIR Access

DIR may be written using the DIR0EXEC register (see Section 5.5.5) or using DIR0 through DIR*n* (DIR*i*) registers (see Section 5.5.4.7).

5.5.4.6 DIR Encoding

The DIR is as long as the widest Xtensa processor instruction configured. It is always at least 24 bits wide, to hold any instruction from the base Xtensa Instruction Set. In processors configured with wide instructions (FLIX), maximum instruction width is between 32 and 128 bits inclusive (register space is reserved for up to 1024 bits). The DIR is accessed as a series of 32-bit OCD registers, DIR0 through DIR*n*, where *n* equals:

n = (max. instruction width) / 32 - 1.

The DIR is accessible as a series of 32-bit registers that contain successive 32-bit portions of DIR in the same order as instructions are stored in target memory.

There are three ways to view the DIR, as depicted in Figure 5–13 and Figure 5–14

- As a single w-bit wide register (24 <= w <= 128). (This integer form is the same as for the processor's internal instruction buffer).
- As a series of bytes in target memory, containing the instruction starting at the first byte, ordered as the target normally fetches and executes it.
- As a series of 32-bit OCD registers, DIR0 through DIR*n*, mapped directly to the target memory view, each register containing four bytes in the target byte order.

Thus, all instructions start in DIR0. If an instruction is 32 bits or less in length, it fits entirely in DIR0. Otherwise it occupies DIR0 and subsequent DIR*i* up to DIR*n* depending on instruction length.

An instruction narrower than DIR is justified at one end of DIR0, according to target endianness (lsb for little-endian, msb for big-endian). Similarly for the last DIR*i* of a wider instruction that is not a multiple of 32 bits in length. The value of padding bytes in that last word (and of any subsequent DIR*i* register) do not matter, although an all-zero value is suggested.

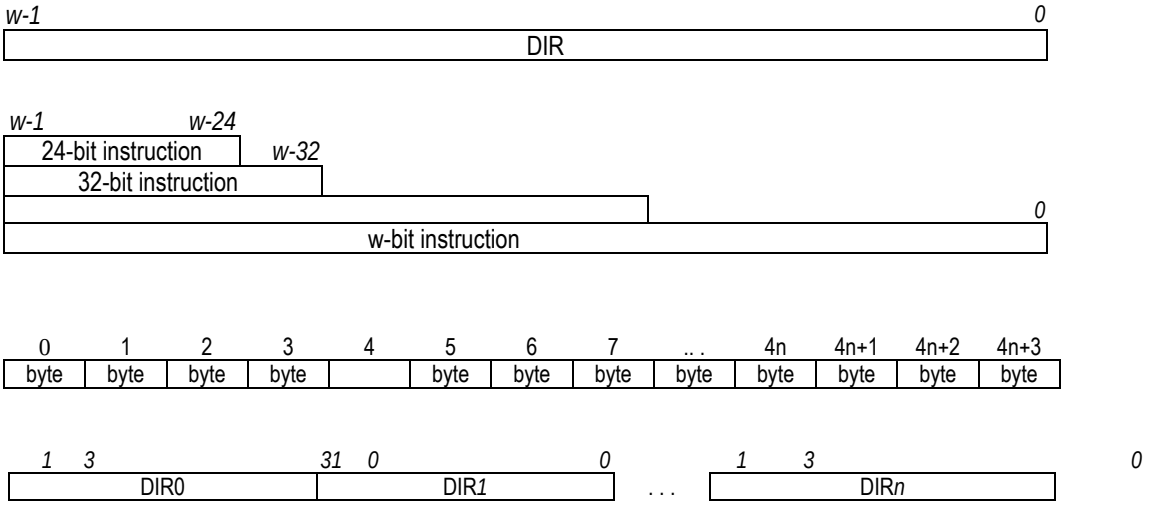


Figure 5–13. Big-Endian Views of DIR

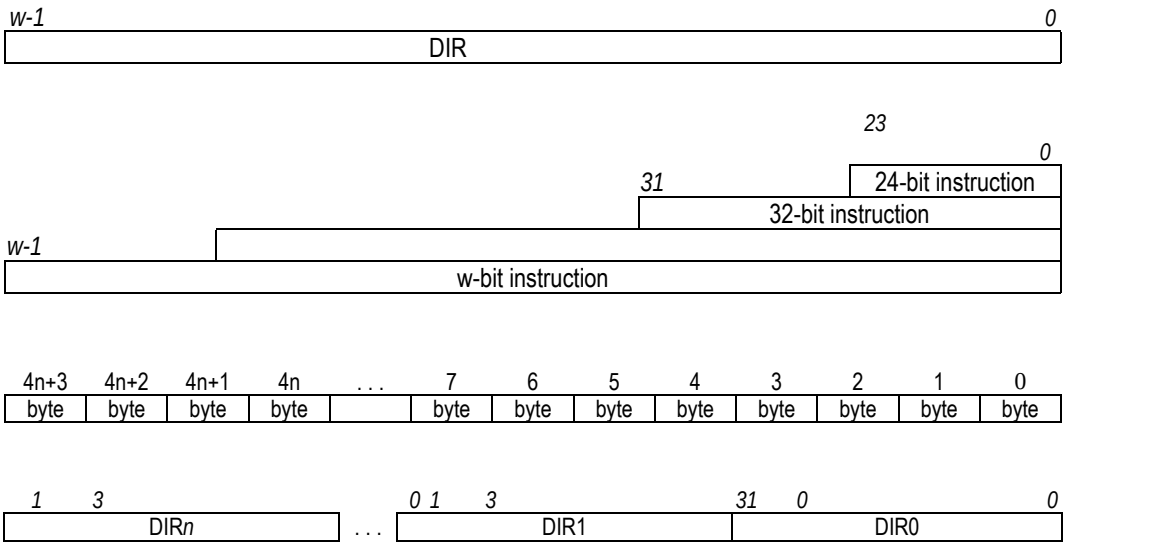


Figure 5–14. Little-Endian Views of DIR

5.5.4.7 DIR0 through DIR*n* (DIR*i*)

The DIR*i* registers are 32-bit chunks of a single DIR register.

Writes to DIR*i* registers are ignored (and the access responds with E=1) if the DSR.ExecException or DSR.ExecOverrun bit is set, or if the processor is not in the *Stopped* state, e.g. if the DSR.Stopped bit is clear.

Otherwise, if the processor is still busy executing an instruction from DIR (DSR.ExecBusy is set), the design assumes an *overrun* condition: the DSR.ExecOverrun bit transitions to being set, and the write is ignored (DIR is not updated). The access responds with E=1 and B=1. In all other cases writes to DIR*i* respond with E=0 and B=0.

Reads of DIR*i* respond with E=0 (DIR*i* is always accessible) and B reflects the value of DSR.ExecBusy.

5.5.5 DIR0EXEC

DIR0EXEC is an alias to DIR0, with the added side-effect of executing the instruction when written. To execute an instruction wider than 32 bits this way, the relevant DIR1 to DIR*n* registers must be written *before* writing to DIR0EXEC. See Section 5.5.4.1 for details of how to request instruction execution.

DIR0EXEC is not accessible over ERI.

Reading DIR0EXEC is equivalent to reading DIR0 register - no side-effect and with the same B and E response.

Writes to DIR0EXEC respond with E = (DSR.ExecOverrun | DSR.ExecException | DSR.ExecBusy | !DSR.Stopped) and B = DSR.ExecBusy. The E bit returned must accurately reflect whether instruction execution was initiated (i.e. same computation of E as to decide whether to initiate execution).

5.5.6 DDREXEC

Alias to DDR, with added side-effect of executing the instruction already in DIR. Typically used with instructions which write the DDR special register. See Section 5.5.4.1 for details of how to request instruction execution.

DDREXEC is not accessible over ERI, nor by “internal” APB accesses (PADDR[31]=0).

The side-effect of executing the instruction in DIR occurs for both reads and writes, with several caveats for APB accesses. For APB reads it must be an external access (APB[31]=1). For APB writes, along with external accesses, it can also be for internal accesses (APB[31]=0) provided they aren't locked (see Section 4.3). To reiterate, any

external agent accessing DDREXEC must be cognizant that this is a read-side-effect register. When writing to DDREXEC, it is important that the value being written to DDR is visible to the instruction in DIR whose execution is initiated. Also, DDR must not be written before a previous DIR instruction might itself update DDR).

When reading from DDREXEC, the value read from DDR must be read after it is ensured that the previous instruction have completed, and thus have written DDR), and must be read before the instruction in DIR being executed can change it.

Table 5–27 lists the main conditions under which access to the registers may be unsuccessful. It is intended to be a summary of the error conditions described under the preceding register sections. Note that this table does not list all possible logic combinations, as that would require an untenably large table.

Table 5–27. Effects of Reading and Writing Registers under Atypical Conditions

Register	DSR.Stopped = 0	DSR.Stopped = 1 and DSR.ExecBusy = 1	DSR.Stopped && !ExecBusy && (ExecExcept || ExecOverrun)
DCR DSR	RD success, E=0 WR success, E=0	RD success, E=0 WR success, E=0	RD success, E=0 WR success, E=0
DDR	RD success, E=0 WR success, E=0	RD success, E=1 WR fail, E=1	RD success, E=1 WR fail, E=1
DDREXEC	Rd succ, Exec fails, E=1 WR and Exec fail, E=1	Rd succ, Exec fails, E=1 WR and Exec fail, E=1	RD success, Exec fails, E=1 WR and Exec fail, E=1
DIR0EXEC	RD success, E=0 WR and Exec fail, E=1	RD success, E=0 WR and Exec fail, E=1	RD success, E=0 WR and Exec fail, E=1
DIR0 - DIR*n*	RD success, E=0 WR fails, E=1	RD success, E=0 WR fails, E=1	RD success, E=0 WR fails, E=1

5.6 Core Instructions for Fast Download and Upload

Two 24-bit core instructions are included when an Xtensa core is configured with the OCD option: SDDR32.P and LDDR32.P. They improve memory upload and download speed.

Typically, the following three instructions are executed in sequence in order to write data to memory:

```
rsr.ddr   a3
s32i      a3, a2, 0
addi      a2, a2, 4
```

This sequence transfers the memory data word from DDR to a general register, stores the data to the memory and then increments the memory address register to point to the next memory location. However, it is simpler and more efficient to use a single instruction that performs all three operations, and need not be continually reloaded into the DIR.

SDDR32.P as is a 32-bit store from DDR to memory at the virtual address contained in address register as. After the store is done, address register as is incremented by 4.

LDDR32.P as is a 32-bit load from memory to the DDR register at the virtual address in address register as. After the load is done, address register as is incremented by 4.

These instructions can be used in combination with DDREXEC: each may be pre-loaded in DIR0, and subsequently executed whenever DDREXEC is accessed to transfer a word from or to memory.

SDDR32.P and LDDR32.P are two of only a few memory reference instructions that can access instruction RAM. See the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for details on these instructions.

5.7 Xtensa Processor Instructions for OCD

This section describes special OCD instructions implemented in the Xtensa processor.

These instructions are only valid for issue from OCD, using the mechanism described in Section 5.2.2. Although the assembler may recognize and assemble them, these instructions cannot be fetched normally (from cache or memory, that is, outside of *Stopped* core state); doing so results in an illegal instruction exception.

5.7.1 RFDO — Return From Debug Operation

The RFDO instruction causes the processor to return from OCD mode (i.e. exception or interrupt that caused entry into *Stopped* state) and transition to the *Running* state. The PC of the instruction that was killed upon entry into *Stopped* state is stored in a special return-PC register that is similar to, but separate from, EPC. The RFDO restores the current PC from this special return-PC register. The processor subsequently resumes fetching instructions normally (i.e., from the cache or memory).

All Control Transfer Instructions (e.g. JX) executed while in OCD will cause the special return PC to be updated. In other words, upon execution of RFDO the core will redirect to the PC set by the CTI and not to the instruction that was killed by the OCD interrupt or exception.

Subsequent debug exceptions transition the core debug state according to the debug exception mechanism described in Section 5.2.

Note: The RFDO instruction takes a single immediate operand which is reserved for future use. In terms of instruction operation, the operand value is immaterial. Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for encoding details.

5.7.2 RFDD — Return From Debug and Dispatch

Note: The RFDD instruction is not implemented in Xtensa NX processors.

The RFDD instruction returns from the debug exception (that caused entry into the *Stopped* state). The current PS is restored from EPS[DEBUGLEVEL] and the current PC is redirected to DEBUGVECTOR. Thus, the execution restarts from DEBUGVECTOR rather than EPC[DEBUGLEVEL]. Further debug exception transition the core following the debug exception mechanism described in Section 5.2.

This feature is used by an OCD-based debugger to transfer handling of an exception to a target-based debugger. All debug exceptions are first trapped by OCD. If the OCD software determines that the exception needs to be handled by the target-based debugger, it first needs to disable the OCD functionality and then executes RFDD. This redirects handling of the exception to a memory based debug handler located at DEBUGVECTOR.

Note: The RFDD instruction takes a single immediate operand which is reserved for future use. In terms of instruction operation, the operand value is immaterial. See the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for encoding details.

5.8 Multicore Debug

The typical SoC of a Cadence customer contains many processor cores and peripherals. To debug software running on a given core is to stop its execution — via a debug interrupt or exception — so as to relinquish control to a debugger running on a host. Problems can occur however, if other processors and peripherals on the SoC continue when this happens. For example, a second processor might run into a timeout while waiting for a response from the first, complicating debug and subsequent restart of the system.

In general, when debugging software running on a given core of a multiprocessor system, it is important that the other cores and peripherals not change the state of the system. This requirement is addressed by synchronous stop and resume. Because the stop/resume information should be communicated as quickly as possible to both cores and peripherals, it is preferable for it to be based on on-chip signals rather than commands through debugger software running on the host.

The Xtensa processor addresses synchronous stop/resume by using the BreakIn and BreakOut interfaces of the core. Figure 5–15 shows that synchronous stop is achieved when the debugger sends a command to stop an individual core, followed by that core broadcasting the debug interrupt information through an on-chip Break network.

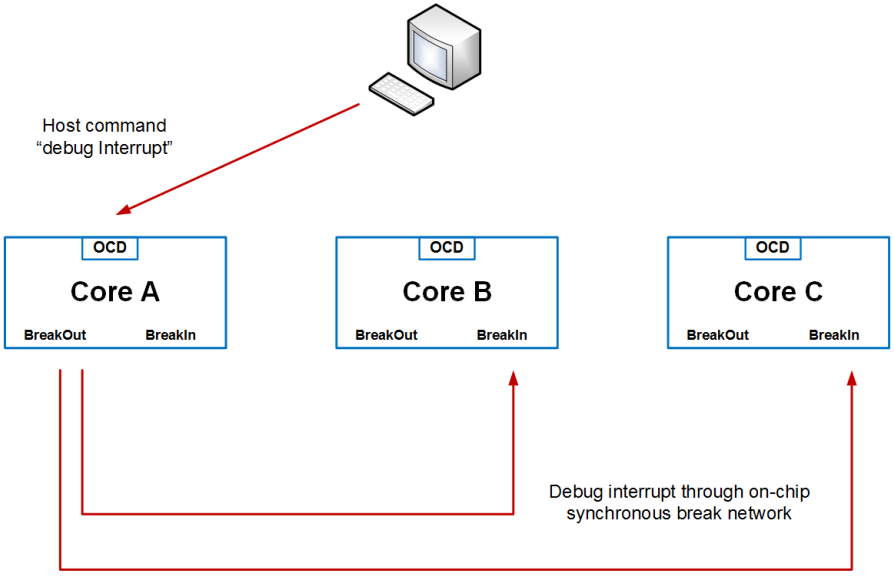


Figure 5–15. Synchronous Stop using Break Network

The `BreakOut` interface is used to communicate a break to other processors when the target processor has entered the *Stopped* state, and the `BreakIn` interface is used by the recipient processor to also enter the *Stopped* state.

The following sections describe the `BreakIn` and `BreakOut` interfaces and provide sample ways of architecting a Break network.

Note that the `BreakIn` and `BreakOut` interfaces are compatible with Arm’s Cross Trigger Interface (CTI) as specified in the *CoreSight Components Technical Reference Manual* Revision H, publicly available from Arm. The subsections below make reference to connection to Cross Trigger Module (CTM) signals for explanatory purposes, but the `BreakIn` and `BreakOut` interfaces can be used independent of (i.e. without any) CoreSight components. In either case, there is an assumption that the signals are asynchronous to CLK, and therefore there is appropriate synchronizing logic inside Xtensa.

5.8.1 BreakIn Interface

BreakIn is an interrupt signal that causes the core to take a Debug interrupt. It has a corresponding acknowledgment signal as shown in Table 5–28. This is to ensure that signals can be transmitted across widely-separated or asynchronous clock domains.

DCR.BreakInEn is used to enable an interrupt due to BreakIn. If enabled, BreakIn results in the processor taking a vectored Debug interrupt if DCR.EnableOCD is not set, or going into *Stopped* state if DCR.EnableOCD is set.

Table 5–28. BreakIn Interface Signals

Name	Bits	Source	Description
BreakIn	1	System e.g. CTM	Take debug Interrupt. Example CTM connection would be from ECTTRIGOUT[0]
BreakInAck	1	OCD	Acknowledge of BreakIn. Example CTM connection would be to ECTTRIGOUTACK[0]

BreakInAck acknowledges that the BreakIn signal was received. It is a level acknowledge that says that a synchronized version of BreakIn is seen by the internal logic of the Debug module. Since it is a simple synchronized reflection of BreakIn, it trails BreakIn by three cycles, and has the same number of assertion cycles.

Note that BreakInAck does not signify that the Debug interrupt was actually taken. If BreakInEn is enabled, and the core is not already in Debug interrupt, BreakIn is qualified inside the Xtensa processor. If these conditions are true, BreakInAck indicates that the Debug exception was taken.

Normally, BreakInEn would be true prior to the arrival of BreakIn, but this need not be always the case. If BreakIn is kept asserted, and subsequently BreakInEn is set, this would also cause the Debug exception.

5.8.2 BreakOut Interface

BreakOut is a signal that communicates that the processor went to the *Stopped* state. It has a corresponding acknowledgment signal as shown in Table 5–29. This is to ensure that signals can be transmitted across widely separated or asynchronous clock domains.

Table 5–29. Breakout Interface Signals

Name	Bits	Source	Description
BreakOut	1	OCD	Debug interrupt taken. Example CTM connection would be to ECTTRIGIN[0].
BreakOutAck	1	System e.g. CTM	Acknowledge of BreakOut. Example CTM connection would be from ECTTRIGINACK[0].
XOCDMode	1	OCD	Processor is in *Stopped* or *Stepping* state.

`BreakOut` assertion requires `DCR.BreakOutEn` to be set.

Assertion of `BreakOutAck` by the CTM or external logic causes the Xtensa processor to deassert `BreakOut`. Due to synchronization flops inside the Debug module, it takes at least three clock cycles for the de-assertion. However, if `BreakOutAck` is tied high, `BreakOut` is guaranteed to be a one-cycle pulse because the high value would have propagated into the core by the time it emerges from reset.

In the absence of `BreakOutAck`, `BreakOut` does not remain indefinitely asserted. Once the core goes out of OCD mode (out of *Stopped* or *Stepping*) it is cleared.

5.8.3 The Break Network

The user needs to develop a Break interconnect network appropriate to their system and this section gives examples.

Typically, one would want `BreakOut` from any processor propagated to `BreakIn` of all processors. To accomplish synchronous stop, using the debugger a user would set up the `DCR` register in each processor to enable OCD and to receive and propagate break interrupts. If any processor hits a break, all processors in the system will break within some bounded number of cycles.

5.8.3.1 Break Interconnect Network Example 1

In this example, a network takes the BreakOut/BreakIn interface signals (total of four signals) and performs two basic functions:

a. For a group of cores belonging to the same synchronization group (a set of cores to be stopped whenever one of them stops) perform logical OR of their `BreakOut` signals and drive `BreakIn` signal of all of them. This ensures all the cores will stop when one of them stops.

b. For the same group of cores, perform logical AND of their `BreakInAck` signals and drive `BreakOutAck` signals of all of them. This ensures that the `BreakOut` of a core entering the *Stopped* state won't get deasserted until the very last core in the group registers it and stops.

Note that avoidance of combinational-only paths in user logic requires that the `BreakOut` of each core **not** be used in OR logic in (a) to drive its own `BreakIn` input and the `BreakInAck` of each core should **not** be used in AND logic in (b) when driving its own `BreakOutAck` signal. This essentially demands for as many OR and AND operations as there are cores in each synchronous group. Alternatively, the break interconnect network signals could be passed through free-running flops thereby avoiding combinational-only paths.

5.8.3.2 Break Interconnect Network Example 2

If the cores belonging to a break-synchronization group run on different (i.e. asynchronous) clocks, clock-domain-crossing logic must be employed in the break interconnect network. This is easy to do because the BreakIn/BreakOut interfaces are designed with such synchronization logic in mind. Figure 5–16 shows an example where the `BreakOut` of any core is transmitted to the `BreakIn` of all cores just as in Example 1.

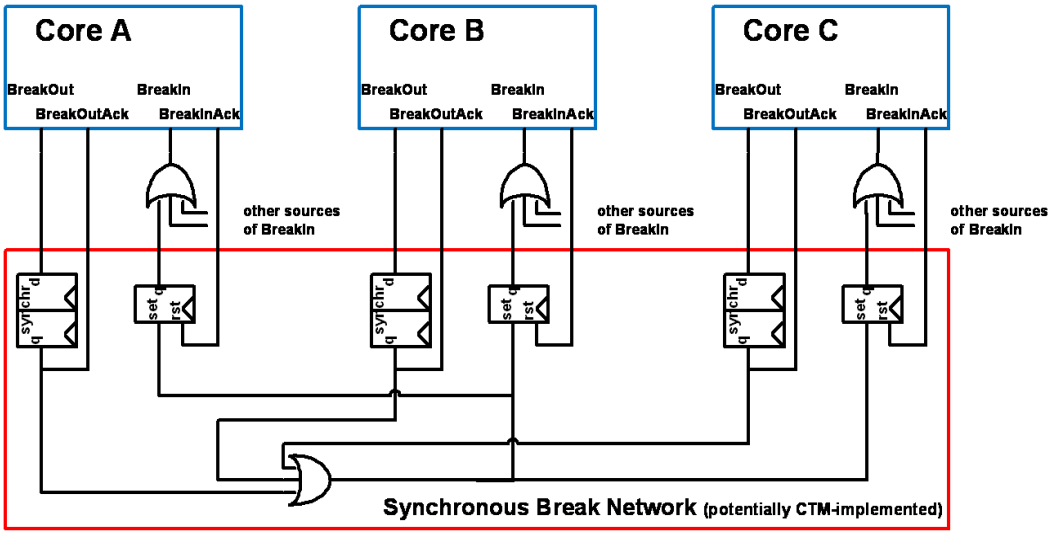


Figure 5–16. Example of Break Interconnect Network

There are certain differences with Example 1 demonstrating the plurality of ways of designing a break interconnect network. The first is that there is no logical AND of the `BreakInAck` signals driving the `BreakOutAck` signals of each core. Instead, the

BreakOuts become cleared within a small number of synchronization cycles. The second is that the BreakIn of a given core has OR'd in other break sources. This allows a particular core to participate in more than one break synchronization group.

The BreakIn/BreakOut interfaces are designed to be compatible with Arm's CTI interface protocol. Therefore, the synchronous break network of Figure 5–16 can easily be mapped on to the Cross Trigger Module (CTM) of Arm's Embedded Cross Trigger (ECT) architecture. For more details, see the CTI documentation referenced at the beginning of this Section 5.8.

5.8.3.3 Break Interconnect Network Example 3

One simplified implementation of the network interconnect is in systems where all the cores run on the same clock, so all BreakOutAck signals can be tied high. In this case, BreakOut signal pulses for just one clock cycle which is sufficient for other cores to register it. As a result, the network interconnect consists of a single OR logic circuitry which takes all the BreakOut signals and drivers all BreakIn signals. The BreakInAck output is simply ignored.

5.8.4 Restarting Processors in a Multiple Processor System

After stopping, it is not possible to resume without debugger intervention — with only a Break network. In other words, the debugger has to connect to all the cores and be in a position to send instructions. Synchronous resume is achieved by sending instructions to each Xtensa pipeline to return from the debug interrupt — as shown in the following figure.

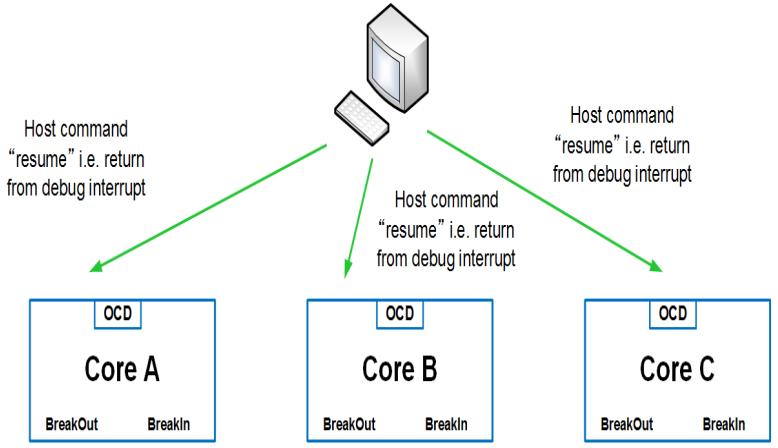


Figure 5–17. Synchronous Resume of a System of Xtensa Cores

In Figure 5–17, all processors are in the *Stopped* state to begin with. To restart the cores at the same time, the debugger executes the `RFDO` instruction on each one. This is accomplished by loading the `DIR` registers of all cores with `RFDO` and then simultaneously issuing the command to execute the instruction. This would be done either through the APB interface or the JTAG interface to the OCD module. Due to differing delays from the debugger to each core, there could be very many cycles of difference in when the cores execute the instruction. With JTAG, this is somewhat mitigated if the TAP controllers of the cores are chained together, but even in this case synchronous restart is only within tens of processor cycles.

The Xtensa OCD Daemon supports this simultaneous restart feature. Refer to Section 6.16 for more details.

5.8.5 DebugStall (for Xtensa LX Processors only)

The method of synchronous stop and resume described in Section 5.8.3 and Section 5.8.4 above involves debugger intervention. This means that the micro-architectural and architectural state of processors that are not immediately the target of debug interest are changed — because they are forced into the *Stopped* state and required to execute at least one instruction.

Also, as mentioned above, the connection of the host to each core is through a probe and then to an on-chip JTAG or APB bus. While synchronous stop through the Break network is reasonably fast (within tens of clock cycles), synchronous resume could be several hundreds of cycles apart due to the delay of sending individual resume commands.

The preceding assumes homogeneous MP cores. A bigger issue is that when cores are heterogeneous, synchronous resume may not happen within a reasonable number of cycles. In fact, to achieve any kind of synchronous resume requires debugger-to-debugger coordination as the example in Figure 5–18 shows.

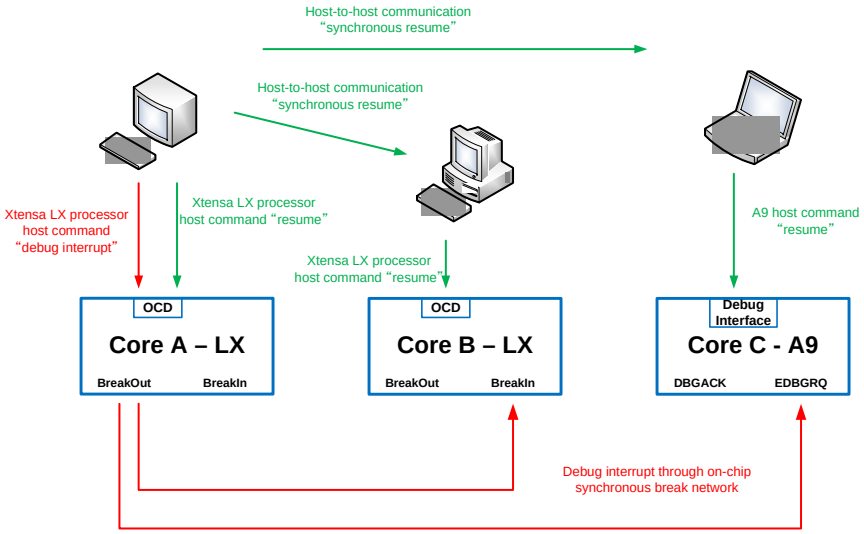


Figure 5–18. Synchronous Stop/Resume with Xtensa LX6, Xtensa LX5 and ARM Cortex-A9

In summary, resume using Break network alone has shortcomings:

- debugger intervention is required and therefore processor state is affected
- it requires central coordination

Particularly with regard to the latter, coordination becomes impractical when multiple processor architectures and other IP blocks are involved. It requires integration of hardware and software across various companies and user systems. The level of standardization required may not be available to the Xtensa user.

A simpler solution uses the stall input pin of a processor, which in the case of Xtensa is RunStall. A debug-aware RunStall, denoted hereinafter as “the DebugStall feature” (or just DebugStall) is used to achieve quicker stop and resume. The idea is simply to stall processors rather than force entry into a state-changing debug mode.

As shown in Figure 5–19, synchronous stop is achieved when debug mode — as indicated by the signal XOCDMode — on the target processor causes the RunStall signal of other processors to be asserted.

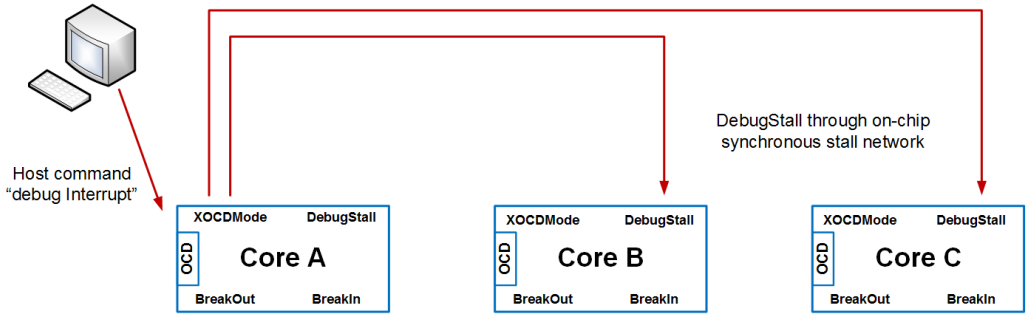


Figure 5–19. Synchronous Stop using Stall Network

This method of synchronous stop is faster than that using the Break network because the RunStall assertion causes the pipeline to stall immediately. With BreakIn, one must wait for the core to retire the instructions in flight and re-steer instruction fetch to the debug vector or engage the OCD logic.

As shown in Figure 5–20 synchronous resume is achieved when exit from debug mode on the target processor causes the RunStall signal of other processors to be deasserted.

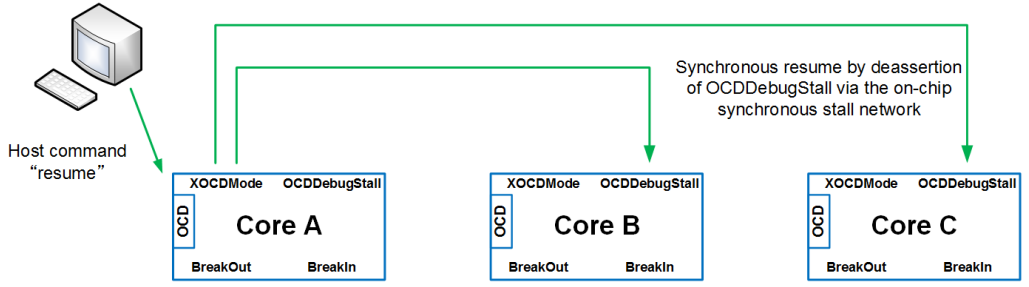


Figure 5–20. Synchronous Resume using Stall Network

The idea then is that a processor that is not the immediate debug target stops quickly and also resumes quickly with minimum disruption to SoC state. However, the DebugStall feature also makes it possible for the other debuggers attached to the other processors to examine the state of their respective targets if they so chose. Crucially, this examination can happen without disrupting the synchronous stop/resume paradigm assumed by the first debugger.

As a corollary, the DebugStall feature makes it possible for symmetric synchronous resume to be possible — no matter what debugger interventions happen during the stall period. Symmetric in this case simply means that the last debugger to cause its target to exit debug mode would initiate synchronous resume in the manner expected by all debuggers. Or more simply, the debugger that caused the synchronous stop in the first place need not be the one to send the final resume command.

In this manner, the DebugStall feature also addresses a separate requirement that when a core is stalled via RunStall, it should be debuggable. A common situation is where a master processor controls initial program loading (and then startup) of Xtensa cores — that are configured with OCD. When (as Cadence recommends) RunStall is asserted by the master to prevent the Xtensa cores from running, these slaves were (in previous Xtensa releases) not accessible by a debugger. Debuggability is an issue not only in this type of multicore initial program load but also when starting up and/or downloading code to a single core under debugger control.

To understand the preceding better, let us walk through the debug scenario of the three core MP system of Figure 5–20 with reference to the three states that a core could be in:

- The *Break* state which is simply that the core is in debug mode. The connected debugger has full access i.e. the ability to issue instructions, look at memory, etc. The Xtensa core does not execute any instructions except those specifically initiated by the debugger. In this state, the XOCDMode output of Xtensa is asserted.
- The *Stalled* state. The Xtensa core is not executing any instructions. In this state, there is no debugger connected.
- The *Running* state. The normal running state of the processor.

These states are shown in Figure 5–21.

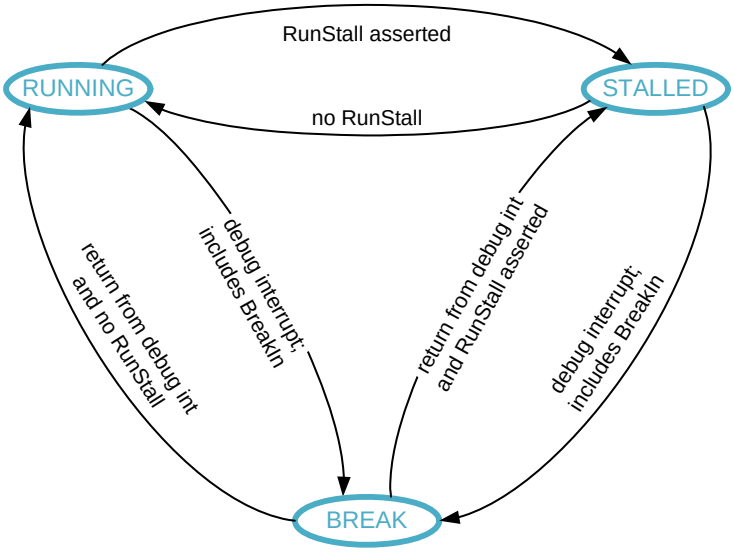


Figure 5–21. State Transitions using both DebugStall and Break Networks

Notice that this is simply an extension of the state machine shown in Figure 5–12 earlier this chapter. To wit, the *Running* states are identical, *Break* encompasses the *Stopped* and *Stepping* states, and the *Stalled* state is new. So in fact, if the use of DebugStall is disabled, i.e. MP debug relies solely on the Break network, the state machine degenerates to that shown in Figure 5–22 below.

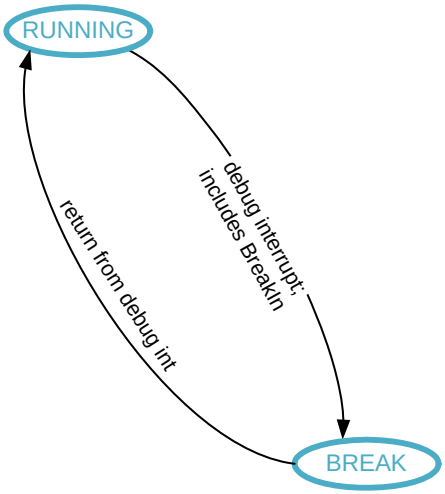


Figure 5–22. State Transitions using the Break Network Alone

Figure 5–21 shows how the state transitions occur. The transitions are based on a debug-mode-qualified RunStall input and detection of a debug interrupt. As mentioned above, a prime cause of the latter is assertion of the `BreakIn` signal. For the purposes of the following discussion however, we assume that `BreakIn` is disabled, and the debug interrupt is caused by a breakpoint, or actively by the debugger issuing a break command to OCD logic by way of JTAG or APB.

The state transitions from *Running* to *Stalled* address the common MP debug scenario as previous sections have alluded to. To wit, the user is interested in debugging SW running on core A. At the breakpoint in question though if other cores continue running, there is an increased potential that the state of the system is changed. It could be sufficiently changed that the debugger is not able to correctly divine system state, memory contents, etc. at the time of the original breakpoint on core A. What is often desired is to minimize all *other* system activity as soon as that breakpoint occurs. A way to do this is to cause all the other cores in the system, in this case B and C, to go to the *Stalled* state.

The transition back to *Running* from *Stalled* allows synchronous resume. Again continuing the scenario above, when core A is taken from *Break* to *Running* by the debugger, the XOCDMode output becomes deasserted. This in turn causes the RunStall inputs to B and C to be deasserted, thereby moving these cores from *Stalled* to *Running*. In other words all three cores go to the *Running* state simultaneously.

Coming back to synchronous stop case, it is possible to move cores B and C from *Stalled* to *Break*. This is possible because while the processor is stalled, the Debug module is still actively running and responds to debug interrupt requests by the host debugger — or indeed other agents. With all three cores in the *Break* state, it is possible to have even more control of the target MP system and to create more calibrated debug scenarios. Synchronous resume is achieved by first taking cores B and C back to the *Stalled* state, and then core A to the *Running* state. Finally, it is possible to enable the `BreakIn` of each core such that synchronous stop immediately causes all cores to go in *Break* state.

5.8.6 DebugStall Hardware and Software Components (for Xtensa LX Processors only)

The RunStall input to Xtensa causes a global stall within the processor (internal signal GlobalStall asserted), freezing execution. The DebugStall feature uses the RunStall mechanism to stall the processor, but prevents the stall when debug activity is occurring. Any debug activity on the core thwarts the stall, so the processor is allowed to execute code:

- when in a hardware debug mode
- when debug software is running (such as a software debug monitor, or while single-stepping)
- when an event or interrupt occurs — or is pending — which would lead to such a debug state

The internal state `DebugMode` captures these conditions, so the necessary condition for GlobalStall to be averted is that Xtensa is in `DebugMode`. Notice that this is not a sufficient condition for GlobalStall to be averted because `DebugStall` (i.e. the `DebugMode` qualified `RunStall`) is OR'd in as one of the GlobalStall sources. If GlobalStall is happening for some other reason e.g. load initiated to unresponsive PIF device, the debugger will find that it is not possible to issue the next instruction.

Conversely, note that the assertion of GlobalStall freezes the processor pipeline, but does not cease all Xtensa activity. As described in Appendix A of the *Xtensa Microprocessor Data Book*, in-progress external transactions (e.g. loads or stores to local memory or PIF, cache-line fills and castouts, TIE-queue transfers, and inbound-PIF requests to instruction or data RAM) will continue during a GlobalStall.

As soon as debug activity ceases (e.g. a GDB user types “continue”), and assuming RunStall is still asserted, the core stalls again. The core resumes — and using the DebugStall feature other cores resume in concert — when RunStall becomes deasserted. Reassertion of RunStall will again stall the core, and a second/subsequent debug event will again unstall the core.

Bits in the DCR and DSR configure and control DebugStall. Specifically, these are `DCR.DebugSwActive`, `DCR.RunStallInEn` and `DCR.DebugModeOutEn` in the debug control register, and `DSR.RunStallInputValue` and `DSR.RunStallToggle` in the debug status register. The functions of these bits are as follows. [Also see section Section 5.5.1 for a description of all DSR bits, and Section 5.5.2 for description of all DCR bits.]

The output pin XOCDMode indicates that the core is in *Stopped* or *Stepping* state as mentioned above, and by virtue of connection to their RunStall input, other cores use this signal to halt their activity. This output is an AND of `DebugMode` and `DebugModeOutEn`.

`DebugSwActive` indicates that the processor is in a user-controlled debug mode. The bit is readable/writable by SW running on Xtensa through the ERI (external register interface) or by an external agent through JTAG or APB. Typically, the external agent would be a debugger running on a host. For the RunStall signal to cause pipeline stall, this bit must be cleared. In other words, the debugger would set this bit in order to force a core to go from the *Stalled* state to the *Break* state so that it could perform debug operations on the core.

While the preceding applies to the target core, `DebugSwActive` also has meaning for the cores that are the recipients of the stall from the target processor. For example, the debugger might wish to do synchronous single stepping where the other processors are continuously held in *Stalled*. Depending on the implementation, synchronous stepping by the debugger might involve momentarily taking the target core out of *Break* and into *Running*. In such a situation, the debugger would set the `DebugSwActive` bit prior to the single step.

`RunStallInEn` masks the RunStall input signal i.e. RunStall is recognized only when it is set. This bit would be cleared in scenarios where the user requires the core to make broad execution progress. For example, core A is in debug mode, and all other cores are stalled because the XOCDMode of core A is connected to all other cores’ RunStall inputs. Now a user connects with a debugger to core B and not only wants to do single stepping, but wants to step over functions i.e. the debugger sets a breakpoint in some removed (i.e. remote) section of the code and does a “continue”. Here the user would want to ignore the RunStall input signal for an extended period of time.

`RunStallInputValue` and `RunStallToggle` are used more for informational purposes with respect to the DebugStall feature. These bits are used by the debugger to respectively sample and confirm the RunStall seen by a core.

There is no configuration parameter that controls the presence of the control and status bits described above. The DebugStall feature is automatically available when the OCD configuration option is selected. There are also no variations of this feature based on other configuration options.

5.8.7 DebugStall Network – aka Synchronous Stall Network (for Xtensa LX Processors only)

In connecting the XOCDMode and RunStall signals in an MP system, the user may have a simple scenario where for synchronous stop it is always the same core that is stopped first, and similarly for synchronous resume it also always *that* core that is resumed first. In this case, XOCDMode output of the core in question is simply broadcast to the RunStall inputs of the other cores.

More typically, the stop-initiating core or resume-initiating core is required to change dynamically. In such cases a broadcast OR network is necessary. As shown in Figure 5–23, a broadcast OR is where all the XOCDMode signals are ORed together, and the resulting output connected to all the RunStall inputs. For the sake of completeness, Figure 5–23 also shows the BreakIn/BreakOut network connecting the same cores. The two networks are slightly different because the BreakIn/BreakOut signals have different characteristics than the XOCDMode/RunStall signals:

- BreakIn/BreakOut are edge signals (cause interrupt in an edge-triggered manner) whereas XOCDMode/RunStall are level signals.
- BreakIn/BreakOut are designed to be compatible with ARM's CTI specification but XOCDMode/RunStall are not.

Note that BreakIn causes a debug interrupt only if it is enabled.

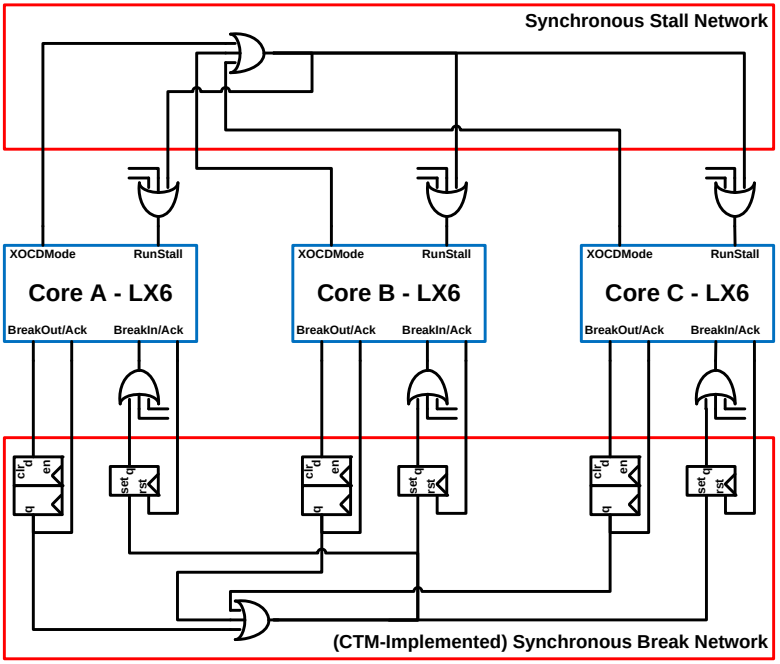


Figure 5–23. Connection Example with Both a DebugStall Network and a Break Network

Of course, Figure 5–23 gives only an example of a DebugStall network. A different structure or topology may be employed if that better suits the system's MP debug requirements. For synchronous stop/resume to work all that is required is that the XOCDMode of the first core to be stopped – or last core to be resumed – is connected to the RunStall of all other cores.

If you are using a complex DebugStall or Break network, check with your support engineer that it is not incompatible with Cadence-provided debug tools. The example network of Figure 5–23 is of course supported by out-of-the-box Xtensa tools.

In the broadcast OR network of Figure 5–23 or indeed any DebugStall network, it is possible for SW to clear `DCR.RunStallInEn` of a given core to have it opt out of synchronous stop/resume temporarily. Notice that this also disables normal functional use of RunStall, as described in Section 5.8.10 below.

5.8.8 BreakIn/Out and DebugStall Connection to an Arm Core (for Xtensa LX Processors only)

How does the user connect the Break and DebugStall network signals in an MP system that includes an Arm core? For detailed information regarding the protocols of stall or break interfaces of the Arm core, refer to the core-specific debug documentation. Following is an example of how one would connect up a system of Xtensa cores and one specific Arm core, the Cortex®-A9.

The Cortex-A9 does not have a pipeline stall in the manner of RunStall. Therefore, it has only the *Running* and *Break* states.

Entry into the *Break* state happens based on a debug exception e.g. breakpoint, or the EDBGRQ signal, which is equivalent to Xtensa's BreakIn. DBGACK is an indication that the core is in *Break* state, so it is equivalent to XOCDMode of Xtensa.

Exit from the *Break* state happens via the debugger or the DBGRESTART pin. If through the DBGRESTART pin, there is an associated acknowledge pin DBGRESTARTED. It is a three-step process:

1. DBGRESTART becomes asserted requesting resume
2. DBGRESTARTED becomes asserted saying "yes, I have exited *Break*" (not "I have resumed")
3. DBGRESTART becomes deasserted causing the processor to start executing instructions

Based on this, the connection to an Xtensa MP system would look as shown in Figure 5–24.

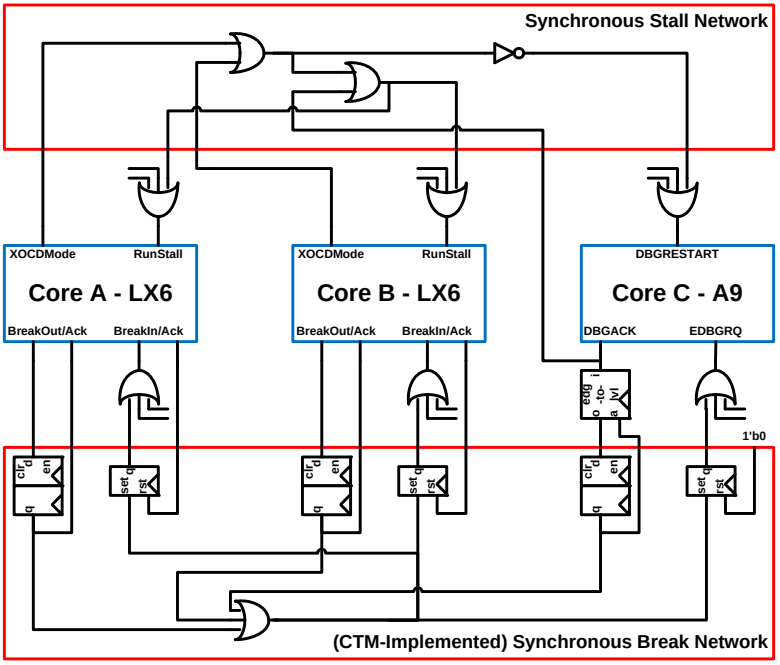


Figure 5–24. DebugStall Network that Includes an Arm Core

In the example of Figure 5–24, when the A9 takes a debug interrupt, it causes the Xtensa cores to be *Stalled* immediately (through the DebugStall network) and then shortly after go into *Break* state. When the Xtensa debugger wants to resume, it restarts the Xtensa cores by executing RFDO on each core. This in turn causes the A9 to resume via its DBGRESTART pin.

5.8.9 Implementation of Break Network Without CTM (for Xtensa LX Processors only)

The network examples in the preceding sections, have all shown use of a CTM. However any logic suffices to construct the Break network, as long as one observes the CTI protocol on the BreakIn/BreakOut interfaces.

Figure 5–25 gives an example of a Break network that does not use a CTM.

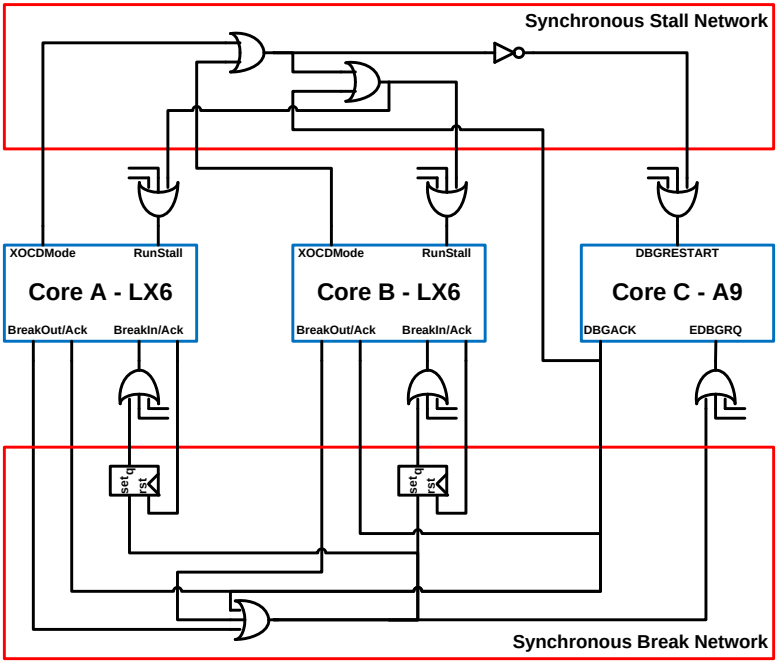


Figure 5–25. Break Network not Implemented using CTM

5.8.10 DebugStall vs. Normal Use of RunStall (for Xtensa LX Processors only)

In the normal case, RunStall has two intended use models, explained in more detail in the *Xtensa Microprocessor Data Book*.

- As a mechanism to initialize local memories using inbound-PIF requests immediately after core reset
- As a means to achieve power reduction that is easier to do than with SW i.e. with WAITI and interrupts

The use of the DebugStall feature does not hinder this normal functioning of RunStall and solely contributes to debuggability of the system — except when a user wishes to use DebugStall and normal RunStall *at the same time*. For example, it is not possible to have one core opt out of the DebugStall-based network at a given point yet at the same time stall on a normal RunStall assertion. This is because there is only one stall input pin, and all sources are ORed outside the core as shown in Figure 5–23. For a given debug session, the user must chose to have the core respond to all sources of RunStall, or none at all.

Alternatively, the user may use a more sophisticated logic to selectively or dynamically deliver different sources of stall to a given core, but this logic would be outside the core and would not be supported by Cadence-delivered tools.

5.8.11 Timing of DebugStall (for Xtensa LX Processors only)

While the goal of the DebugStall feature is synchronous stop and resume for MP debug, it is important to understand that stopping and restarting will not be instantaneous. For example

- There will be latency in the synchronous stall network which is spread across the SoC. Figure 5–26 below shows an example of such a system.
- Inside each Xtensa, the RunStall input is flopped prior to use.
- Setting/clearing of `DebugSwActive` or `DebugModeOutEn` will be through JTAG or APB, with potentially lengthy and differing delay from debugger to core.

While there is not a bounded time by which a processor will stop or resume, engaging the right steps will cause the MP network as a whole to stop or resume in a manner faster than is possible with BreakIn/Out.

Cadence-supplied debug tools will work with uneven delays in either the Break network or the Stall network. For example, they ensure that stopping is not initiated prior to resume correctly propagating — and similarly resuming is not initiated prior to stop correctly propagating — through the entire MP network.

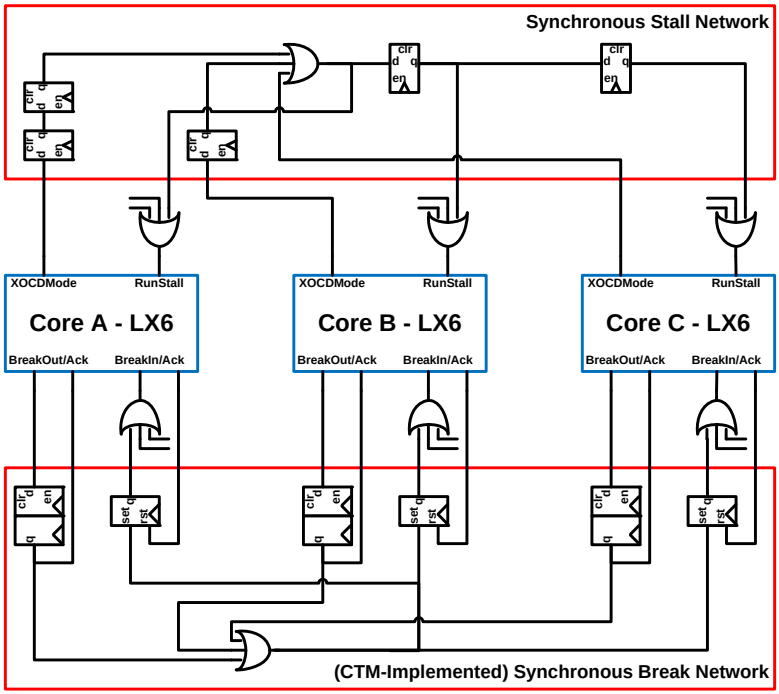


Figure 5–26. DebugStall Network with Uneven Stall Delay

5.8.12 Reset-Time use of DebugStall (for Xtensa LX Processors only)

DebugStall-related logic resides in the Debug module, which is reset by the DReset input of Xtensa (or by software reset through the DebugReset bit of the PWRCTL register). This is separate from the BReset input (with similar SW-controlled reset) which resets the Xtensa core alone. A system power-on reset would normally cause both resets to be asserted whereas a reset of the processor core alone would be done by BReset assertion.

When DReset is asserted i.e. when debug functionality is reset, `DebugSwActive` is cleared and `RunStallInEn` is set. In other words, DReset is the reset signal used for the `DebugSwActive` and `RunStallInEn` flops.

Separately, `DebugSwActive` is also cleared, and `RunStallInEn` also set, when the core is reset. This is done by using the processor reset — BReset — in a synchronous fashion i.e. as a data input.

In short, whether upon system power-on reset or processor-only reset, DebugSwActive is always cleared and RunStallInEn always set. This is to accommodate all situations where a master processor or other agent wishes to stall the Xtensa processor from reset. In this scenario, it is important to honor the incoming RunStall.

5.9 Core and Debug Reset

With regard to the interface between the OCD logic and the processor, special consideration must be given to reset cases. The background of this is that OCD logic is reset by DReset signal whereas the core is reset by BReset.

5.9.1 Core Reset Effect on OCD and Processor

Upon reset, the processor returns to the *Running* state unless the OCDHaltOnReset signal is asserted (Section 5.9.3). The external world can see the reset state of the processor through the CoreWasReset bit of the PWRSTAT register in the Access Port.

In addition, the debug logic performs the right steps to maintain control to the external world:

Most DSR fields maintain their value until they are explicitly cleared by the debugger. This includes ExecDone, ExecException, ExecOverrun, {Core|Host}{Wrote|Read}DDR, DebugPend* and DebugInt* signals.

Other DSR bits directly reflect the state of the processor, so these will be automatically cleared. This includes ExecBusy and Stopped.

XOCDMode is cleared because it is a direct reflection of the DSR.Stopped bit. BreakOut will not be cleared; however it is under direct control of BreakOutAck: the former is cleared if the latter is asserted.

The other registers all maintain their prior values — DCR, DDR, DIR, etc.

5.9.2 Debug Reset Effect on OCD and Processor

If Debug logic is reset while the core is outside of the debugger's control then there is no impact on the core.

If Debug logic is reset while debugging the core, all OCD-related state that is directly controllable by the debugger will be reset. This includes DCR, DDR, DIR and most DSR fields. Only DSR.Stopped and the XOCDMode signal will retain their values, because they are a direct reflection of the state of the processor. So, when a debugger first attaches, it must always check the state of DSR.Stopped. When proceeding with a new

debug session, it may be that the core is already in *Stopped* state even without having issued a debug interrupt. If so, the debugger is then free to execute a return (e.g. RFDO) or proceed issuing new instructions.

5.9.3 OCDHaltOnReset

The Debug module provides an external interface to force processor into the *Stopped* state when the processor comes out of reset. `OCDHaltOnReset` is a 1-bit input signal to OCD and must be driven by external logic. If the signal is high at processor reset, OCD is automatically enabled (`DCR.EnableOCD` is set) followed by the Debug Interrupt to the processor (`DCR.DebugInterrupt` is set). This feature is useful for debugging or to download code using OCD from a separate agent.

Refer to Section 3.3 for the rules of timing that must be met when using `OCDHaltOnReset`.

5.9.4 Debug and Core Simultaneously Emerge from Reset

This is the basic mode of operation where the core begins execution in *Running* state.

The exception to this is if `OCDHaltOnReset` is asserted. The reset de-assertion sequence required is described in Section 5.9.3.

6. Using the Xtensa OCD Daemon (XOCD)

XOCD allows direct control over the connected processor cores through OCD (On-Chip Debug) and does not require a stub running on the target. Thus, the target core can be stopped at any time to examine the state of the core, memory, and program, and therefore provides greater target visibility. This also allows you to debug reset sequences, interrupts, and exceptions. XOCD implements a GDB compatible stub interface and works with any debugger that supports the GDB remote protocol over a TCP/IP connection.

Note: XOCD version 13 is compatible with GDB and TRAX tools version 10 and later versions.

The Cadence OCD solution accesses the on-chip processors' debug logic, OCD, TRAX, Performance Monitors, and Power registers, collectively named Xtensa Debug Module. Accessing the Xtensa Debug Module (XDM), in turn, drives the processor states.

6.1 Debug Module Access Interfaces

The XOCD supports three types of connections to the XDM:

- JTAG
- Serial Wire (SWD)
- Direct access to the XDM (also called APB-direct connections)

All three connection types partially assume the organization of the internal hardware; the following subsections contain the details.

6.1.1 JTAG Interface

The Cadence XOCD can use a JTAG interface to control the internal Test Access Port (TAP) controller with access to processor states. The JTAG interface is usually exported through an OnCE-compatible connector (refer to Section 2.3.10) and requires an external JTAG probe (sometimes called a scan controller) to access the TAP controller from a PC. JTAG probes typically provide common interfaces to the PC, such as Ethernet, USB, or a combination thereof. Because JTAG interfaces (TAPs) are designed such that they can be chained together, users may connect any combination of Xtensa processors and other TAP controllers serially in a chain.

In the example in Figure 6–27, the JTAG probe is connected to the USB port of the PC workstation and to the 14-pin OnCE connector of the target board. `xt-gdb` connects to XOCD over TCP/IP. Thus `xt-gdb` can run on either the same machine as XOCD, or on a separate workstation or server.

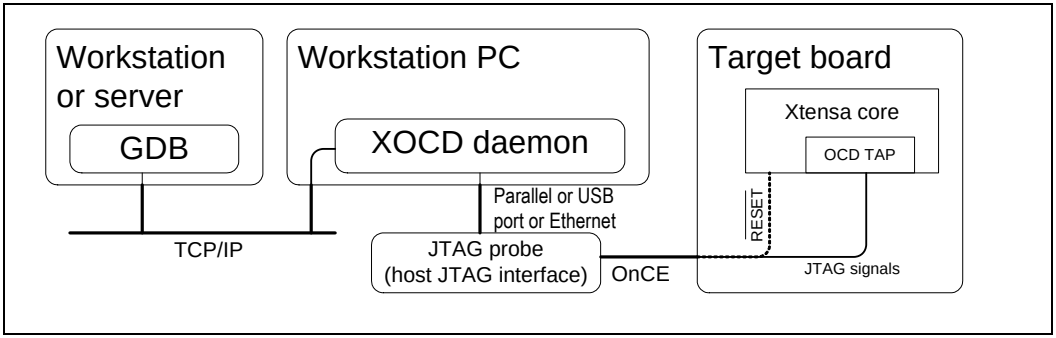


Figure 6–27. Connecting **`xt-gdb`** to the Target Board Using XOCD

Note: Throughout this chapter, `xt-gdb` and the XT-AV60 evaluation board are used as examples for debugger and target, but Xtensa OCD debugging works just as well in any SoC environment and with other debuggers supporting the GDB Remote Protocol.

6.1.2 Serial Wire Debug Interface

XOCD supports Arm's Serial Wire Debug (SWD) interface with probes that are able to drive SWD. SWD is a 2-pin interface developed for debugging in pin-constrained situations. Currently, the use of SWD assumes that the Arm Debug Access Port logic (DAP) drives the XDM module (that is, the DAP is the module residing between the SWD pins and the APB bus where the XDM is memory mapped).

6.1.3 XDM-Direct Interface

When mapped into the APB space, the XDM module is accessible through a set of APB mapped registers. Thus, any logic that has access to the APB space can drive the XDM module, and in turn, the processor. XOCD can be configured to issue only the XDM reads and writes on its external API interface. You are then able to implement the libraries XOCD can use to drive the custom user hardware logic that takes care of routing the requests to the APB bus where XDM resides.

Note: To add support for such a custom probe, consult the SDK delivered with XOCD.

6.2 Host and Probe Support

XOCD directly supports the probes listed in Table 6–30. See Section 6.9 through Section 6.11 for details on how to configure XOCD to use these probes.

Note: XOCD is available as a 64-bit executable only, in both Linux and Windows.

Table 6–30. XOCD Host and JTAG Probe Support Matrix

Vendor	Probe	Windows 7 64-bit	Linux RHEL5/6 64-bit
Arm®	DSTREAM (USB/Ethernet)	Yes	Yes
Cadence	XT-ML605, XT-KC705 daughtercards (USB)	Yes	Yes
Tin Can Tools	Flyswatter2 (USB)	Yes	Yes
SEGGER®	J-Link (USB / Ethernet)	Yes	Yes
Cadence	VDEBUG virtual probe (JTAG & APB)	Yes	Yes

Table 6–31. Example Third-Party OCD-Based Debuggers and Probes

Vendor	Debugger	Probe(s)
Sohwa & Sophia Technologies	Watchpoint	EJ-SCATT (for Xtensa LX to Xtensa LX4)
Yokogawa Digital Computer	microVIEW-PLUS	adviceLUNA (for Xtensa LX to Xtensa LX4), adviceLUNA II (for Xtensa LX to Xtensa LX5)

6.3 Installing the Xtensa OCD Daemon (XOCD) on Windows

The Xtensa OCD Daemon auto-installer is included in the Xplorer installer. For details about installing Xtensa Xplorer, refer to the *Xtensa Development Tools Installation Guide*. Refer to the “Downloading and Installing Xtensa Tools within Xplorer” section in the *Xtensa Development Tools Installation Guide* for specific details about downloading the Xtensa OCD Daemon using Xtensa Xplorer.

If downloaded separately, start the XOCD installer for Windows by double-clicking it or running it from a command prompt. It is named:

```
xt-ocd-<version>-windows-installer.exe
```

In this document, we refer to the XOCD installation directory as <*XOCD_ROOT*>. The installer prompts for its location, which defaults to:

```
\Program Files[2]\Tensilica\Xtensa OCD Daemon <version>.
```

If all options are selected, the installer places the following files in <*XOCD_ROOT*>:

- `ReadMe.txt`: Introduction
- `xt-ocd.exe`: Xtensa OCD Daemon (XOCD)
- `topology.xml`: XOCD configuration file
- `topology-example.xml`: Example `topology.xml` file
- `xt-ocd-log.txt`: Default log file (created on first invocation)
- `uninstall.exe`: Uninstaller
- `Uninstall Xtensa OCD Daemon <version>`: Shortcut to uninstaller
- `FTDI/*`: FTDI (FT2232 chip) drivers and files
- `modules/*`: Xtensa OCD modules (dynamic. loadable)
- `rv/*`: DSTREAM probe drivers and files
- `sdk/*`: JTAG probe driver example sources

Note: You need administrator privileges if FTDI USB probe support is selected during installation. This allows the installer to install corresponding USB driver files where Windows can find them.

6.3.1 Installing FTDI USB Probe Support

Many development and debug tools use FTDI for their USB interface, for example the ML605 and KC705 FPGA systems, or the Flyswatter2 from Tin Can Tools. We recommend that FTDI-based USB probes be disconnected from the host PC before installing XOCD, and connected only after installation or when prompted to do so. This helps ensure they are properly recognized using the newly installed drivers.

When installing support for FTDI-based USB probes, Windows may prompt several steps as part of installing the driver, as follows.

- If a Found New Hardware Wizard message appears, asking whether to connect to Windows Update, select No (“No, not this time”).
- If prompted, select to install the software automatically, not from a specific location.
- Windows may prompt multiple (up to four) times that the software for this hardware has not passed Windows Logo testing. Select “Continue Anyway” in all cases.
- If Windows prompts to reboot to complete the installation, select to do so later, so as to allow the XOCD installation to complete. Then, reboot Windows manually.

2. On 64-bit Windows systems, this may be "`Program Files (x86)`" instead of "`Program Files`".

6.3.2 Installing SEGGER J-Link Probe Support

XOCD automatically provides support for SEGGER J-Link probes. However, XOCD does not install the probe driver, nor the corresponding USB drivers. To install these, you need to obtain the SEGGER J-Link Software and Documentation Pack from http://www.segger.com/jlink-software.html (or http://www.segger.com/j-link-older-versions.html) and follow the provided documentation.

Note: Currently, XOCD cannot search for the J-Link drivers installation directory on Windows. After installing the drivers obtained from the SEGGER website, copy `Jlink_x64.dll` to `<XOCD_ROOT>/JLink` directory.

Note: Cadence tests only some J-Link probe driver updates with XOCD. The last J-Link probe driver version tested was 6.20 for Windows and Linux. A later driver may be necessary if that one is no longer available.

6.3.2.1 Installing J-Link Drivers for Linux

When installing J-Link drivers on Linux, you must create a symbolic link within the XOCD directory pointing to the J-Link probe driver shared library. The following example illustrates this: XOCD expects the driver (named *libjlinkarm.so.6*) to be located in the *modules* directory, whereas the driver is actually in the J-Link installation directory. For example:

ln -s <jlink-install-dir>libjlinkarm.so.6 <xocd-install-dir>/modules/libjlinkarm.so.6

6.3.3 Uninstalling the Xtensa OCD Daemon

To remove the Xtensa OCD Daemon software, you can either:

- Select **Uninstall** in the Start menu entry for Xtensa OCD Daemon, or
- From the control panel (**Start** → **Settings** → **Control Panel**), double-click **Add/Remove Programs**, and select Xtensa OCD Daemon.

Note: This will not uninstall the `mac_mot.sys` driver if you have installed it.

6.3.4 Adding Support for a Third-Party Probe

It is possible to develop a custom probe driver, as a dynamically loadable library used by XOCD according to its topology file. XOCD is a layered software; one of the layers is the probe API, which is used to communicate with all included probe drivers (delivered as dynamic libraries). The same API is used to implement the driver for any custom probes. XOCD installation files provide the environment (makefiles, source files examples, and

instructions) for you to produce your own probe driver dynamic library. For details see the README and other source files under the `/sdk/` subdirectory where XOCD is installed.

6.4 Installing the Xtensa OCD Daemon (XOCD) on Linux

The Xtensa OCD Daemon auto-installer is included in the Xplorer installer. For details about installing Xtensa Xplorer, refer to the *Xtensa Development Tools Installation Guide*. Refer to the "Downloading and Installing Xtensa Tools within Xplorer" section in the *Xtensa Development Tools Installation Guide* for specific details about downloading the Xtensa OCD Daemon using Xtensa Xplorer.

The XOCD self-extracting installer for Linux is run by simply executing it, for example:

```
./xt-ocd-<version>-linux-installer
```

In this document, we refer to the XOCD installation directory as <*XOCD_ROOT*>. The installer prompts for its location, which defaults to `/opt/Tensilica/xocd-<version>`. The installer places the following files in <*XOCD_ROOT*>:

- `xt-ocd` Xtensa OCD Daemon (XOCD)
- `topology.xml` XOCD configuration file
- `topology-example.xml` Example `topology.xml` file
- `ReadMe.txt` Introduction
- `FTDI/*` FTDI (FT2232 chip) drivers and files
- `modules/*` Xtensa OCD modules (dynamic. loadable)
- `misc/*` Installer / uninstaller scripts
- `rv/*` DSTREAM probe drivers and files
- `sdk/*` JTAG probe driver example sources
- `uninstall` Uninstaller

For the FTDI driver, the installer also creates files here:

- `/etc/udev/rules.d/*-xocd-ft2232*.rules`

To uninstall the Xtensa OCD Daemon, run <*XOCD_ROOT*>`/uninstall`.

Note: If the installer encounters an unexpected error, the latest installer log file under `/tmp/bitrock_installer_*****.log` may provide some useful information.

6.5 Installing the Xtensa OCD Daemon (XOCD) on OS-X

Currently, XOCD does not support any probes under the OS-X. Instead, XOCD is bundled as a zipped (*tgz)* file incorporating the main program, required libraries, probe SDK, and the topology file examples. From here, you can add your probes support using the SDK.

6.6 Running and Connecting to XOCD

Before starting XOCD, make sure that the JTAG probe is connected to the target and that the target is powered up and ready. Refer to Section 6.2 for additional information about supported JTAG probes and options.

6.6.1 Running XOCD on Windows

XOCD can be invoked from a command shell or by its default shortcut in the Windows Start menu (Start → Programs → Xtensa OCD Daemon → OCD Daemon).

You can edit the Start menu shortcut to change command-line parameters, listed in Section 6.6.3. By default, the shortcut passes these parameters: `-dTD=20 -T 10.`

6.6.2 Running XOCD on Linux

The Xtensa OCD Daemon can be invoked as <*XOCD_ROOT*>`/xt-ocd`. For example, assuming <*XOCD_ROOT*> was added to the PATH environment variable:

```
xt-ocd -dTD=20 -l xt-ocd-log.txt
```

The OCD Daemon may output status messages as it runs, so it is usually invoked in its own shell window.

6.6.3 XOCD Command-Line Parameters

The Xtensa OCD Daemon (`xt-ocd`) accepts the following optional command-line parameters (Note: some parameters can be specified using either their short name or the long name. The short name requires a single "**-**" prefix and allows "=" when assigning the parameter value, as well as blank space.):

- **`-c`**, **`--config=<`***`configuration-file`***`>`**

  Use the specified XOCD configuration file instead of looking for `topology.xml` in the current and XOCD installation directories (in that order).

- `-r`, **`--rvconfig=<`***`realview-configuration-file`***`>`**

For use with Arm DSTREAM probes only (refer to Section 6.11). Use the specified realview configuration file (normally with an.`sdf` extension) instead of looking for `rv-conf.sdf` in the current and XOCD installation directories.

- **`-d`**`<layer>[<thread>][=<`*`level`*`>]`

  Enables XOCD logging. XOCD will provide additional information useful for debugging software or hardware problems. The required <*layer*> argument must be either `T`, `D`, or `TD`. It specifies the set of modules to log (T=target, D=device). The optional <*thread*> argument is a decimal integer that specifies the application thread to log. Each `<application>` element in the topology file is an application thread (see Section 6.7.5), numbered sequentially from zero in the order defined. The default is to log all threads. For a given thread, the first matching `-d` option has effect, so `-d` for a specific thread must precede `-d` for all threads. The optional debug <*level*> argument specifies the verbosity level. The value can be between 0 (no debugging) and 99 (most verbose). Default is 0, or 40 if the `-d` option is provided without specifying <*level*>. Refer to “Debugging Messages” on page 108 for more information about this option.

- **`-l, --log=<`***`filename`***`>`**

  Use the specified log file for logging information instead of `stderr`.

- `-`**`T, --trace=<`***`level`***`>`**

  Enable the XOCD internal function call tracing at level <*level*>, ranging from 0 (no trace) to 99 (most detailed trace). This provides potentially useful information for Cadence to debug software or hardware problems.

- `-`**`L, --Log=<`***`filename`***`>`**

  Use the specified file for internal tracing information instead of `stderr`.

- **`-I, --init=<`***`filename`***`>`**

  Supplied file contains arbitrary JTAG commands to execute when XOCD starts (refer to Section 6.13 “XOCD Initialization Script”).

- **`-R, --rst=<`***`filename`***`>`**

  Supplied file contains arbitrary JTAG commands to execute after each TAP Reset is executed. Refer to Section 6.13 “XOCD Initialization Script”.

- **`-J, --scans=<`***`filename`***`>`**

  Records all JTAG or SWD commands that are sent to the probe and the data received back. Occurrences of system or TAP reset are also recorded. See Section 6.14 for details.

- **`--trst`**

  Executes JTAG reset using JTRST pin upon XOCD start.

- **`--srst`**

  Executes system reset using a SRST pin upon XOCD start.

- **`--query`**`=scanchain`

  Queries the scan chain and print result. If used with `--trst` switch, the command also tries to determine the IDCODE of each device.

- **`--query`**`=probes`

  Queries for all the probes which have support for this feature implemented in the XOCD. The supported probes are listed in `probes.xml`, which is also the topology file used when XOCD starts under this option. The `probes.xml` file is organized as a multi-core jtag chain, with each core having its own probe. It is the user's responsibility to remove a probe (and the corresponding core and the TAP) if the drivers for that probe are not installed. **Note**: Some probes require specific procedures that must be executed to properly install them. For example, FTDI chip-based probes (see Section 6.9) require re-plugging a probe after XOCD is installed to finish the FTDI driver installation.

  `--override`

  Forces interruption of the core during operations that are normally not interruptible by the debug interrupt, even if it means that the state of the system becomes corrupted. The default state is off.

6.6.4 Connecting to XOCD

The debugger and XOCD can run either locally on the same machine or remotely on different computers connected to the same network. XOCD listens on the TCP port specified in the topology file for the first Xtensa core for any incoming connections (by default, it is 20000). Subsequent cores can be accessed through successive port numbers (20001, 20002, etc.) or can be assigned different ports. Only one instance of the daemon can listen on the same TCP port at any given time, so attempting to start multiple instances of the OCD daemon on the same machine will normally fail unless given unique `topology.xml` files with customized port numbers to allow otherwise.

Note: If the machine running XOCD has a firewall in place, and you are connecting from a different machine, you may need to configure the firewall to allow incoming traffic over the TCP ports used by XOCD according to its XOCD topology file (for example, ports 20000 through 20000+*n*).

Refer to the *GNU Debugger User's Guide* for GDB or the Xplorer online help about the procedure to establish a connection to XOCD from `xt-gdb` or Xplorer, respectively. In Xplorer, the host name and port number may be entered when creating a new launch configuration of the type "Attach to Xtensa GDB Port" under the "Run->Debug" menu. In `xt-gdb`, use '`target remote <xocd-host>:20000`'. If successful, you will be able to debug the target processor as usual. Note that the processor could have been in an arbitrary state when you attached to it via OCD, so you may need to set up some basic register state (such as CACHEATTR or MMU registers, PS, etc.) before loading and run-

ning a new program. If the reset signal on the target's OnCE-style connector is properly routed to the processor's reset signal, the GDB `reset` command is the simplest way to reset the processor state.

6.6.5 Debugging Errors

Whenever a debug operation issued from GDB fails, XOCD sends a coded error to the GDB indicating the error condition. Following are the existing error codes:

- E00: Reserved for general internal errors.
- E01: Unknown/uncategorized cause.
- E02: Malformed packet (GDB Remote Protocol packet).
- E03: Error accessing the memory. Beside errors as a result of general GDB Remote Protocol commands to access the memory (e.g., 'm', 'M' and 'X') this also includes program download errors and memory access errors during TIE registers access ('Qxtreg' and 'qxtreg' GDB commands).
- E04: Processor disconnected unexpectedly.
- E05: Arguments to a GDB Remote Protocol command are not valid.
- E06: Error accessing a register as a result of general GDB Remote Protocol commands to access registers ('p' or '.P').
- E07: Reserved for very specific internal errors, such as failures to execute GDB `monitor` commands (Section 6.19).

Besides sending the error response to the GDB, all debugging errors are sent to the command-line interface (`stderr`) and to the log file, if specified with the `-l <file>` option. Thus, the XOCD output or the log file provide more details on the actual cause of a debug error.

6.6.6 Debugging Messages

XOCD can be instructed to print additional debugging information with the -d <*layer*>[=<*level*>] option. This option requires that you specify the layer or module that should print additional debugging information and a level to set the verbosity of the messages. The layer argument can be either the target (`T`), device (`D`), or a combination of both (`TD`). The target layer provides an interface to `xt-gdb`. If debugging is enabled, it prints all commands received from `xt-gdb` and its responses. The device layer prints the generated stream of instructions it is executing on the target processor.

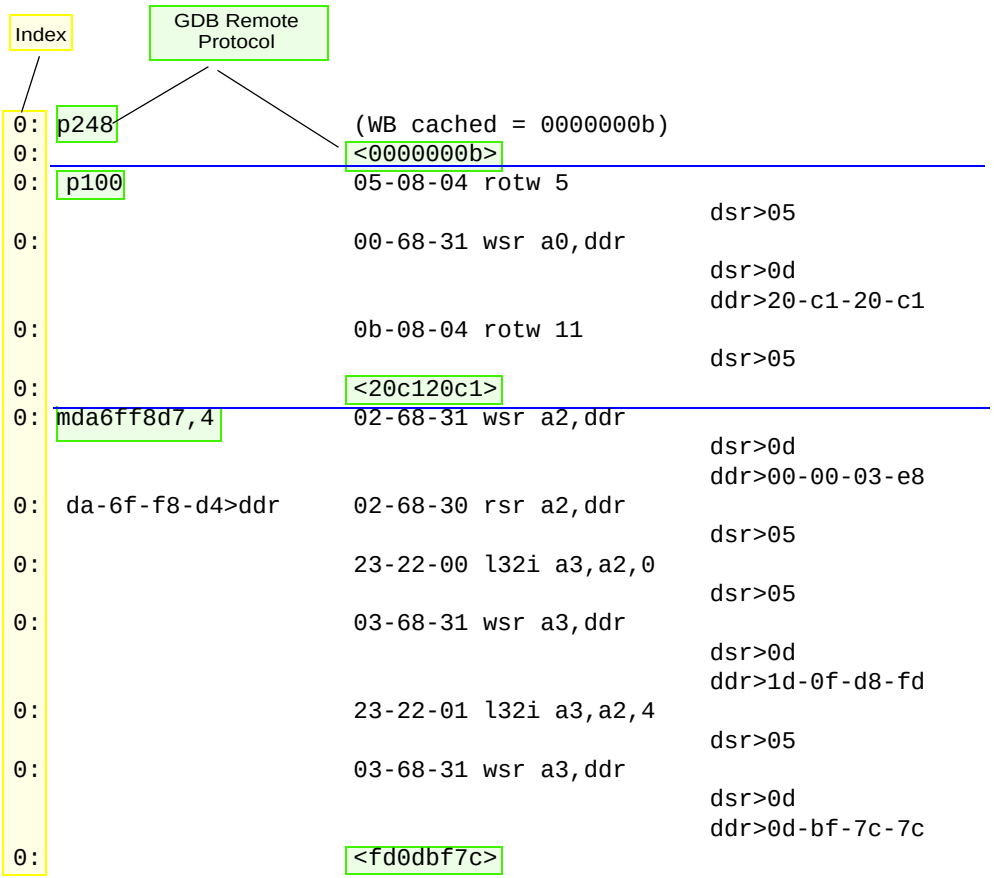
The level can be a value between 0 and 99, with a default value of 40. Useful values are described in the following table:

Table 6–32. Debug Level

Level	Description
30	Minimal output. Only show connection or target details.
40 (default)	Default output. Print protocol details and the instructions executed on the target.
50	Verbose debugging information. Print additional information.

All debugging messages are sent to the command line interface (`stderr`) unless a log file was specified with the `-l <file>` option.

The following is an example of the output generated by XOCD with `"-d TD=40"`:



The *index*: column indicates the target or device instance in an MP environment. Commands sent by the debugger are printed next to the index and the responses are printed inside angle brackets (< and >). The remaining text shows the activity of the OCD on the target as a result of the GDB commands. In the preceding example, p*register-number* is a request from GDB to read the value of the specified register, and m*address*,*length* is a command to read a number of bytes (length) from the specified memory location (address).

The first command (p248) is a request to return the value of WINDOWBASE. In this case, XOCD has already read and stored the value of this register into an internal cache. Therefore, XOCD can directly copy the value <0000000b> from its cache without executing any instruction on the target. For performance reasons, XOCD uses a register cache to keep copies for frequently used registers, such as address registers, WINDOWBASE, and WINDOWSTART. When a cached register is requested by the debugger, XOCD indicates this with: (register_name CACHED = value).

The second command (p100) is a request for register ar0. Because XOCD only has access to the current window registers (a0 to a15), it must rotate WINDOWBASE to the correct base, read the corresponding register for ar0 (which is now the same as a0 after the rotation), and revert the rotation. The value returned is <20c120c1>.

All instructions executed on the target are displayed in form of the opcode followed by the mnemonic. In the preceding example, '05-08-04' is the opcode for the instruction 'rotw 5' in target byte order. Whenever data is exchanged between XOCD and the Xtensa processor through the DDR register, XOCD indicates this by printing 'xx-xx-xx-xx>ddr' for setting the DDR register (printed in front of the instruction) and 'ddr>xx-xx-xx-xx' for reading the DDR register (printed after the instruction). 'dsr>XX' shows that XOCD has read the debug status register (DSR) and also includes the value. The DSR indicates whether a command has completed, an exception has occurred during execution, and/or whether the DDR register has been updated by the processor.

6.7 The XOCD Topology File

The XOCD topology file describes the layout of the target system's JTAG scan chain, as well as parameters related to the JTAG probe, GDB agent, and TRAX. The intent of the topology file is to give XOCD enough knowledge of the scan chain that the user can debug software running on Xtensa processors.

Because the JTAG chain layout is outside the Xtensa processor(s), it is not automatically configured by the Xtensa Processor Generator. The default topology file describes a simple scan chain that consists of a single Xtensa processor. If the scan chain is composed of more than one Xtensa processor or contains other JTAG devices, the topology file must be edited accordingly.

By default, XOCD looks for a file named `topology.xml` in the current directory, and if not found there, it looks in the directory where XOCD was installed. See Section 6.6.3.

The following example topology file describes a single Xtensa processor with TRAX on a scan chain accessed using a Flyswatter2 JTAG probe.

```
<configuration>
  <!-- This line selects the JTAG probe and its parameters: -->
  <controller id='Controller0' module='ft2232' probe='flyswatter2' speed='10MHz'
  />
  <!-- This line describes the Xtensa processor: -->
  <driver id='XtensaDriver0' module='xtensa' step-intr='mask,stepover,setps' />
  <!-- This line describes the TRAX unit: -->
  <driver id='TraxDriver0'   module='trax' />
  <!-- This section describes the JTAG chain connected to probe Controller0 -->
  <chain controller='Controller0'>
    <tap id='TAP0' irwidth='5' />
  </chain>
  <!-- This section describes TAP0 as supporting Xtensa processor + TRAX -->
  <system module='jtag'>
    <component id='Component0' tap='TAP0' config='trax' />
  </system>
  <device id='Xtensa0' component='Component0' driver='XtensaDriver0' />
  <device id='Trax0'   component='Component0' driver='TraxDriver0' />
  <!-- This section describes the GDB remote target agent: -->
  <application id='GDBStub' module='gdbstub' port='20000'>
    <target device='Xtensa0' />
  </application>
  <!-- This section describes the TRAX target agent: -->
  <application id='TraxApp' module='traxapp' port='11444'>
    <target device='Trax0' />
  </application>
</configuration>
```

The following describes each XML element and attribute. TRAX related details are described in Section 6.7.9. Most element instances refer to each other by name. Figure 6–28 depicts this visually for reference: each arrow links two matching names.

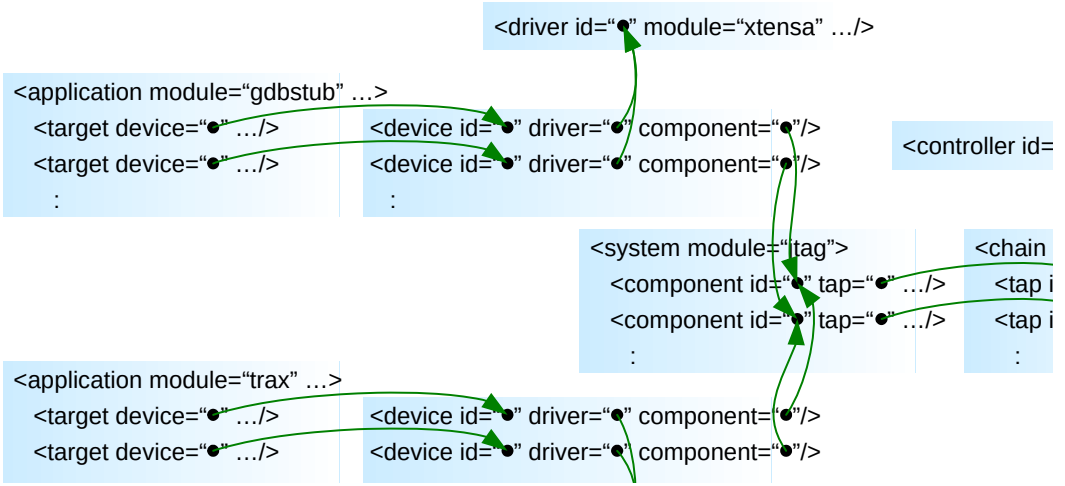


Figure 6–28. Naming Relationships Among Topology File Elements

6.7.1 `controller` Element

The `controller` element defines the physical JTAG probe (sometimes referred to as a scan controller). Following are its attributes:

- `id` — Unique identifier (name) for this probe. This is matched against the `chain` element's `controller` attribute value.
- `module` — Name of the loadable library (without the `.dll` or `.so` suffix) for this probe's driver. For example, `ft2232` (Refer to Section 6.9).
- `type` — Target board connection type. Can be either 'jtag', 'swd' or 'xdm'. Type 'xdm' is used for setups where XOCD is to execute memory-mapped reads/writes to the XDM module. XOCD controller SDK can be used to implement the user-defined communication with the user system/hardware that is set up to execute Debug APB transactions. This argument is only required for J-link and TCP probes.
- (others...) — Other attributes are specific to the probe driver. Refer to Section 6.9.

6.7.2 chain and tap Elements

The chain element describes the JTAG chain connected to a JTAG probe, as a sequence of tap elements that each describe a TAP controller along the scan chain. The first tap element describes the TAP controller whose TDI is connected to the data output of the JTAG probe, and whose TDO is connected to the TDI of the next TAP controller, if any (refer to Section 2.3.9). Following is the only attribute of the chain element:

- controller — Name of the JTAG probe (controller element id attribute) connected to this JTAG chain.

Attributes of the tap element (sub-element of chain) include:

- id — Unique identifier (name) for this JTAG TAP controller. This is matched against the component sub-element's tap attribute.
- irwidth — Width of the TAP controller's JTAG instruction register (IR) in bits. Default is 5. For Cadence cores, this must be 5.
- bypass — Encoding (irwidth bits wide) of the TAP controller's BYPASS instruction. Default is all bits set, according to JTAG standard.
- bypasswidth — Width of the TAP controller's BYPASS data register. Default is 1 (one), according to JTAG standard.

6.7.3 system and component Elements

The system element provides an additional abstraction layer and lists all components of the system. For JTAG systems, the components represent the TAP controllers and, therefore, refer back to the corresponding tap element defined earlier. Only components that map to the Xtensa processor TAP controllers must be listed here; other TAP controllers can be omitted. Attributes of the system element include:

- module — Name of the loadable library (without the .dll or .so suffix) for this system abstraction. Always "jtag".

Attributes of the component element (sub-element of system) include:

- id — Unique identifier (name) for this component. This is matched against the device element's component attribute.
- tap — Name of the TAP controller (tap sub-element id attribute).
- config — Type of TAP controller. One of "tensilica" or "trax" for the Xtensa processor TAP controllers. Must be "trax" when a TRAX unit is connected, which involves a different TAP controller that can connect to the TRAX trace compressor. Otherwise is usually "tensilica". In practice, because the TAP controller used for TRAX is a strict superset of the normal one, it is acceptable to set this attribute to "trax" even in the absence of TRAX hardware or TRAX TAP.

There must be only one `system` element in the topology file regardless of the number of cores or chains described in the topology (see Section 6.7.7 and Section 6.7.9 for examples).

6.7.4 `driver` and `device` Elements

The `driver` and `device` elements create a new device instance for a component. The `device` element is referenced by the application and itself references `component` and `driver` elements.

Attributes of the `driver` element include:

- `id` — Unique identifier (name) for this driver. This is matched against the device element's driver attribute value.
- `module` — Name of the loadable library (without the .dll or .so suffix) for this driver. Usually one of xtensa or trax.
- `step-intr` — (`xtensa` drivers only) Comma-separated option keywords for single-stepping. Currently, this attribute must always be `"mask,stepover,steps"`.
- `debug` — (xtensa drivers only) Comma-separated option keywords for debugging certain XOCD or JTAG issues. Can be one or more of (default is none of these):
  `"verify"` - Wait for completion of every processor instruction fed over JTAG, resulting in significant slow-down but may allow seeing effects of problems more precisely in detailed log files.
  `"forcepower"` - Keep the power to the core after resuming its execution. Prevents core from entering PSO.
  `"noload"` - disables memory write optimizations during the program download.
- `reset` — (`xtensa` driver only) Comma-separated option keywords for controlling whether the system reset, issued by GDB, will also reset the additional logic specified with this option. Keywords are one or more of `"jtag"` (reset the logic driven by JTRST JTAG signal) and `"debug"` (reset Debug logic — logic in the Debug reset domain. **Note:** By default, GDB 'reset' command doesn't issue system reset. To configure GDB reset behavior see '`monitor reset-default`' command in Section 6.19.

- `inst-verify` (`xtensa` driver only) Choose which OCD-fed instructions to verify (wait for their completion). Without this option, no instructions are verified. Keyword `"all"` enables verification of all instructions. Keyword `"mem"` enables verification of memory accessing instructions only. Keyword `"memretry"` first tries all instructions within a single GDB command without verification, and if there is a failure (there always is a final check), the entire GDB command is repeated with `"mem"` verification setting.
- `inst-wait-msec` (`xtensa` driver only) When verifying completion of OCD-fed instructions, poll for given number of milliseconds. If the instruction does not complete in time, an error is thrown.

Attributes of the `device` element include:

- `id` Unique identifier (name) for this device. This is matched against the `target` sub-element's `device` attribute value.
- `component` Name of component (`component` sub-element `id` attribute).
- `xdm-offset` (APB only). Selects the XDM base address in the APB memory space. The attribute is to be used with probes other than DSTREAM.
- `xdm-id` For XDM-Direct probes, either `xdm-id` or `xdm-offset` can be used to specify the address (location) of the XDM in the APB space.`xdm-offset` is typically used. The DSTREAM topology files uses `xdm-id` (refer to Section 6.7.4).
- `dap` Specify dap=`'1'` if the Arm DAP is used to access the core.
- `ap-sel` (APB only). Specifies the DAP access port (a MEM-AP, e.g. APB-AP, AHB-AP, AXI-AP) to which XDM is attached. APB-AP is the access port "1", and is the only AP verified to work with XOCD.
- `driver` Name of driver (`driver` element `id` attribute).

6.7.5 `application` and `target` Elements

Finally, the `application` element configures the top-level XOCD applications, such as the GDB target agent and TRAX control. All targets devices controlled through the application are listed here as `target` elements with a reference to a `device` element. Attributes of the `application` element include:

- `id` Unique identifier (name) for this application. Conventionally `"GDBStub"` for the GDB debug agent, and `"TraxApp"` for the TRAX control agent.

- `module` Name of the loadable library (without the `.dll` or `.so` suffix) for this application's driver. This is `"gdbstub"` for the GDB debug agent, and `"traxapp"` for the TRAX control agent.
- `port` TCP/IP port number for accessing this application. For the GDB debug agent, this is a starting port number: it is used by the first listed GDB target, and is incremented by 1 for each subsequent target. For TRAX control, a single port is used to manage all TRAX compressor units.
- `buffer-size` (GDB debug agent only) Size of GDB command buffer. Default is 16384 (16 KB).

Attributes of the `target` element (sub-element of `application`) include:

- `device` Name of the target device (`device` element `id` attribute).
- `sync-group` (GDB debug agent only) Synchronous group number. Targets of the same synchronous group are assumed to have their debug break-in and break-out signals mutually connected as described in Section 6.16, thus enabling synchronous debugging as described in the same section. Default is 0; if left unchanged for all targets, they are all made part of the same synchronous group, and thus assumed interconnected for synchronous debugging. To completely disable this feature, assign a unique `sync-group` value to each target. See also Section 6.16.1.
- `loose-sync` (GDB debug agent only) If selected ('1'), the GDB target is not resumed together with other cores within the same JTAG cycle. Instead, the final resume command for that target is sent separately. This option allows for synchronous resume in a system with cores implementing different resume mechanisms. Default is '0'.
- `memwidth` (GDB debug agent only). `memwidth='<value>'`: <value> can be 4 or 5; the default is 4. `memwidth='4'` causes XOCD to translate all debugger memory requests into only 32-bit load/store instructions. `memwidth='5'` causes XOCD to translate 1-byte memory requests into only 8-bit load/store instructions, and memory requests of greater than 1 byte into 32-bit load/store instructions. This is useful if the core is connected to some memory-mapped devices that can only accept 4-byte requests and different devices that can only accept 1-byte requests.

6.7.6 Describing the Chain Topology

XOCD supports any scan chain topology that adheres to the IEEE JTAG specification. The topology is described in the `chain` element of the topology file. All TAP controllers (Xtensa processors and other IEEE JTAG compliant TAP controllers) connected to the JTAG probe must be listed here. The topology of the scan chain is implicitly defined by the order of the `tap` elements, where the first TAP element describes the TAP controller closest to the TDI of the JTAG probe.

Adding TAP controllers to the scan chain simply requires you to add another `tap` element to the scan chain with a unique identification. The attribute `irwidth` describes the width of the TAP Instruction Register. For the Xtensa processors, set this value to 5.

```
<chain controller='Controller0'>
   <tap id='TAP0' irwidth='5' />
   <tap id='TAP1' irwidth='5' />
   <tap id='TAP2' irwidth='6' />
</chain>
```

6.7.7 Chain Topologies with Multiple Xtensa Processors

When adding more Xtensa processors to the scan chain, it is also necessary to add a `component`, `controller`, `device`, and `target` element for each additional processor to the appropriate sections. Each element must be assigned a unique name and include a reference to the corresponding element.

```
<configuration>
 <controller id='Controller0' module='ft2232' probe='flyswatter2'
 speed='10MHz' />
 <driver id='XtensaDriver0' module='xtensa'
         step-intr='mask,stepover,setps' />
 <chain controller='Controller0'>
    <tap id='TAP0' irwidth='5' />
    <tap id='TAP1' irwidth='5' />
    <tap id='TAP2' irwidth='6' />
    <tap id='TAP3' irwidth='5' />
 </chain>
 <system module='jtag'>
   <component id='Component0' tap='TAP0' config='Tensilica' />
   <component id='Component1' tap='TAP3' config='Tensilica' />
 </system>
 <device id='Xtensa0' driver='XtensaDriver0' component='Component0' />
 <device id='Xtensa1' driver='XtensaDriver0' component='Component1' />
 <application id='GDB0' module='gdbstub'>
   <target device='Xtensa0' />
   <target device='Xtensa1' />
 </application>
</configuration>
```

The bold lines in this example demonstrate the changes necessary to support a second Xtensa processor. TAP0 and TAP3 represent two Xtensa processors and are referenced by the corresponding component element. TAP1 and TAP2 are other devices on the JTAG chain (not Xtensa processors), for illustration purposes only. Note that all identification names must be unique across the whole topology file.

Note: Currently, XOCD supports only one GDB target agent and all the Xtensa cores (`target` elements) must be listed under the corresponding `application` element.

See also Section 6.16.1 for a description of how to extend the topology file to support synchronous debugging among multiple processors.

6.7.8 Using Multiple Probes

XOCD does not currently support multiple probes. You would need to start one XOCD per used probe.

6.7.9 Editing the Topology File for TRAX

The presence of any TRAX unit must be reflected in the topology file. TRAX support and topology are specified in the topology file as follows:

1. Specify the TRAX driver module after other <*driver .../*> lines:

   ```
   <driver id="TraxDriver0" module="trax" />
   ```

2. Specify all TRAX compressors using a `device` line for each instance as follows. The `id` parameter must be unique for each instance. The `component` parameter must match the corresponding TAP component in the `system` section. The `driver` parameter is constant:

   ```
   <device id="Trax0" component="Component0" driver="TraxDriver0" />
   ```

3. Add the following lines that define the TRAX application module after other similar <*application ...*> ... <*/application*> sections (the `port` parameter is optional). Specify a <*target .../*> line for each TRAX compressor. The `device` parameter must match the corresponding <*device*> line's `id` parameter (as described in Step 2 above).

   ```
   <application id="TraxApp" module="traxapp" port='11444'>
       <target device='Trax0' />
   </application>
   ```

The standard and default port number for the TRAX service is 11444. The `xt-traxcmd` tool, described in Chapter 11, uses this port number by default. To use a different port number, specify it in both the topology file and `xt-traxcmd` arguments.

6.8 Debugging Xtensa Processors in the CoreSight Environment

XOCD supports debugging of Xtensa processors attached to the Arm CoreSight Access Port (APB) using any of the probes in Table 6–30.

Debugging using DSTREAM probes is described in Section 6.15. For all other probes, XOCD requires that you supply `dap='1'` switch as part of the `device` line in the topology file (as described in Section 6.7.4), while the topology file needs to set the base APB address of all the cores (using the `xdm-offset` argument, described in Section 6.7.4).

6.8.1 Debugging using a Serial Wire Debug 2-pin Interface

XOCD also includes the Serial Wire Debug (SWD) driver, which can reach the Arm DAP connected to the host using the Debug Serial Wire protocol.

SWD is currently supported by DSTREAM and J-Link probes only. DSTREAM automatically handles this type of the connection (see Section 6.15). For J-Link probes, the SWD is selected using the type switch in the controller line in the topology file (as described in Section 6.10).

6.9 FTDI Based-Probes – ML605, KC705, Flyswatter2, JTAGkey-2

Probes based on the FTDI series of USB JTAG chips (www.ftdichip.com) are configured by specifying `module='ft2232'` and associated attributes in the XOCD topology file's `controller` element. This provides support for specific probes based on the FT2232 series of chips by FTDI. This currently includes the onboard probe in the XT-ML605 and XT-KC705 daughterboards, and Flyswatter2 probes from Tin Can Tools (www.tincantools.com).

Note: Tin Can Tools offers an adapter (Arm20OnCE14) to connect a probe with the Arm 20-pin connector to a board using the 14-pin OnCE-style connector (refer to Section 2.3.11).

The `probe` attribute selects the specific type of JTAG probe:

- `probe='ML605'` XT-ML605 daughterboard's onboard probe
- `probe='KC705'` XT-KC705 daughterboard's onboard probe
- `probe='flyswatter2'` Tin Can Tools Flyswatter2
- `probe='C232HM'` FTDI C232HM-DDHSL-0 USB to MPSSE cable

The optional `usbser`, `speed`, `clkdivisor`, `vid`, and `pid` attributes are described in Table 6–33.

The following example describes a minimal configuration for the XT-ML605:

```
<controller id='Controller0' module='ft2232' probe='ML605' />
```

Note: On Linux hosts, starting XOCD may issue a system reset pulse on selected Flyswatter2 probes, ML605 and KC705 daughtercards. In addition, on Linux hosts avoid running multiple instances of XOCD that each access one or more FTDI-based probes.

Table 6–33 describes all attributes specific to FTDI-based probes.

Table 6–33. FTDI-Based Probe Attributes

Attribute	Description
`probe`	Describes the probe type. Allowed values: `ML605`, `KC705`, `flyswatter2`
`speed`	JTAG max clock frequency in Hz. May also be specified in kHz or MHz with the corresponding suffix (e.g. speed="15MHz"). See below. Note that the probe uses a clock divisor and can only generate corresponding frequencies (seeTable 6–34). The driver uses the closest frequency equal to or smaller than the one requested. For example, speed="20 MHz" results in an actual frequency of 15 MHz.
`clkdivisor`	JTAG max clock frequency expressed as a JTAG clock divisor (see also Table 6–34). Default is '1'. Mutually exclusive with the speed attribute (use one or the other).
`usbser`	Probe serial number string. On the XT-ML605 and KC705 daughterboards, this has the form "ML605-*NNNN*" or "KC705-*NNNN*" where *NNNN* is the serial number printed on the Cadence property sticker affixed on the daughterboard. Notes: ▪ Flyswatter2, as shipped, has a fixed serial number ("FS20000" or similar). As it cannot be distinguished, only one such probe is supported on a given machine. ▪ If only one FTDI-based JTAG USB probe is connected to the host, this attribute is optional. ▪ If a selected probe isn't found, XOCD displays the list of FTDI USB probes it could find on the machine, along with their serial numbers. See below.

The `speed` and `clkdivisor` attributes both result in a clock divisor value that selects the JTAG probe's clock (JTCK) frequency. Table 6–34 shows the JTAG clock frequencies of some selected `speed` and `clkdivisor` values for the indicated probes. Also shown in italics are the corresponding minimum CPU clock frequencies supported (that keep JTAG clocked at less than 0.3 times the CPU clock frequency, although 1/4 of the CPU clock is a recommended safer ratio.

Table 6–34. Speed Attributes for the ML605, KC705, and Flyswatter2 (FT2232H)

clkdivisor	speed	minimum CPU clock
0	30 MHz	*100 MHz* [1]
1	15 MHz	*50 MHz*
2	10 MHz	*33 MHz*
3	7.5 MHz	*25 MHz*
9	3 MHz	*10 MHz*
n = 0 - 65535	30 / (*n*+1) MHz	*100 / (n+1) MHz*

1. Not supported on the ML605 board.

For an XT-ML605 board with a processor running at 50 MHz, for example, use `speed='15 MHz'` or `clkdivisor='1'` (the default).

6.9.1 FTDI Probes on Linux

XOCD uses the D2XX driver from FTDI to access different FTDI based probes. This driver is unable to access the probe if the Virtual Port (VCP) driver, included in different Linux distributions, is already using the probe (this is true only if the probe uses standard VID (Vendor ID) and PID (Product ID). It is your responsibility to unload the VCP driver (named *ftdi_sio*) before trying to use the probe, by using the Linux command *rmmod ftdio_sio.* The VCP driver is loaded after the probe is attached so the command only needs to be executed when the probe is attached, not before each XOCD use.

The unloading of the VCP driver has an effect on other applications that are using the same driver to access other FTDI devices. For example, serial communication through UART will be broken. To avoid unloading the VCP driver, the FTDI chip used for JTAG access should be reprogrammed to use non-standard Vendor ID (VID) or Product ID (PID). By default, the VCP driver uses standard VID (0x403) and PID(0x6010). Because JTAGKey-2 probes use non-standard VID and PID, they are not required to unload the VCP driver. For all other FTDI based probes, the VID and PID can be programmed into the chip. If using such a probe, the XOCD topology file must set `vid` and `pid` parameters accordingly (refer to Table 6–33).

6.10 SEGGER J-Link Probes

J-Link probes are configured by specifying `module='jlink'` and associated attributes in the XOCD topology file's `controller` element. Both the JTAG interface and Serial Wire Debug (Arm CoreSight) are supported.

The `speed` attribute, and one of the mutually exclusive `usbser` and `ipaddr` attributes, described in Table 6–35, are required.

To determine or modify a probe's IP address or serial number, use the *J-Link Configurator* tool included with SEGGER J-Link software and documentation pack. Alternatively, if you know the IP address, you can use the probe's web server. Refer to the J-Link probe package documentation for details.

The following example describes a minimal configuration for the J-Link probe attached to the USB:

```
<controller id='Ctrl0' module='jlink' type='jtag' usbser='{J-Link
Serial Number}'  speed='{JTCK in Hz}'/>
```

and for the J-Link probe accessed over Ethernet (using IP address):

```
<controller id='Ctrl0' module='jlink' type='jtag' ipaddr='N.N.N.N'
     speed='{JTCK in Hz}'/>
```

Table 6–35 describes all attributes specific to J-Link probes.

Table 6–35. J-Link Probe Attributes

Attribute	Description
`usbser`	Select J-Link probe to use over USB using its serial number. Cannot be combined with the `ipaddr` attribute.
`ipaddr`	Select J-Link probe to use over Ethernet using its IP address, in the form `X.X.X.X` or as a DNS name. Cannot be combined with the `usbser` attribute.
`speed`	Desired JTCK clock frequency, in Hz. J-Link selects the closest (lower or equal) available frequency, and XOCD displays the actual frequency used. This is a required parameter.
`type`	Type of the probe. Either '`jtag`' or '`swd`'

6.10.1 J-Link Support for the Serial Wire Debug Interface

Currently, Serial Wire Debug is implemented to support debugging of Xtensa processors behind the Arm DAP, in the Arm CoreSight environment. Along with the use of the `xdm-offset` argument in the topology file (Section 6.7.4), use of SWD requires the `type='swd'` switch on the controller line of the topology file.

6.11 Arm DSTREAM Probes

Arm DSTREAM probes support debugging of Xtensa cores attached to an Access Port (AXI-AP, AHB-AP or APB-AP) of the CoreSight Debug Access Port (DAP).

DSTREAM probes also allow attaching to the target system using the Arm Serial Wire Debug (SWD) interface, instead of JTAG. SWD is an interface consisting of two pins only, allowing for reduced pin count dedicated for debugging purposes.

XOCD communicates with DSTREAM probes using the Remote Device Debug Interface (RDDI) protocol. The required ARMRDDI libraries are included as a part of the XOCD installer.

6.11.1 Arm Platform Configuration File (.sdf)

Note: *Valid as of DS-5 version v5.29.1*

The Arm Platform Configuration Editor within Arm's DS-5 debugger lets the user configure the system connectivity, including the base address of each Xtensa core Debug Module withing the APB space, the index of the Access Port where Xtensa cores reside, the TCP or USB settings of the used DSTREAM probe, and so on. The Editor output is a configuration file (.sdf) used by XOCD.

For details on the Platform Configuration Editor please refer to the DS-5 debugger documentation.

An Xtensa core (with or without TRAX) is accessed through its Debug module, visible as a 16KB block in the Debug APB space. To access the Debug module, a DSTREAM access template named *CSREG* is used. This template accesses an arbitrary 4KB block in the APB space. Thus, to access all XDM registers, four CSREG templates must be added by the Platform Configuration Wizard, to cover each Xtensa core. The CSREG template, which is an Arm generic template, is available in any DSTREAM probe firmware supporting CoreSight.

When adding each CSREG template, you must set the template base address; that is, the location of the 4KB APB space they cover. For the first of each core's four templates, the base address is the offset of the debug module in the APB space, while the other three templates' addresses are simply consecutive, each at a 4KB offset from the last.

Note: All four CSREG templates are assigned a numerical ID named *RDDI ID*. These IDs must be consecutive for each core, as the XOCD topology file depends on them for proper operation (it needs the ID of the first template for each core). The assigned ID can be obtained from the Device Table tab in Platform Configuration, under "RDDI ID".

The CSREG template is dependent on the use of the Arm DAP. It must be added under the CSMEMAP template (available under *AccessPort* devices) which is to be added under ARMCS-DP template (available under *DebugPort* devices).

Note: The CSMEMAP template must have its base address and AP Type selected.

An example configuration files is installed with XOCD under the `rv` subdirectory. It is configured to access two Xtensa cores, at addresses *0x10000* and *0x20000*:

- `rvc-CSREGx8.sdf` includes two Xtensa cores (with RDDI IDs 3 and 7).

This configuration file look in the Platform Configuration Editor is shown in Figure 6–29.

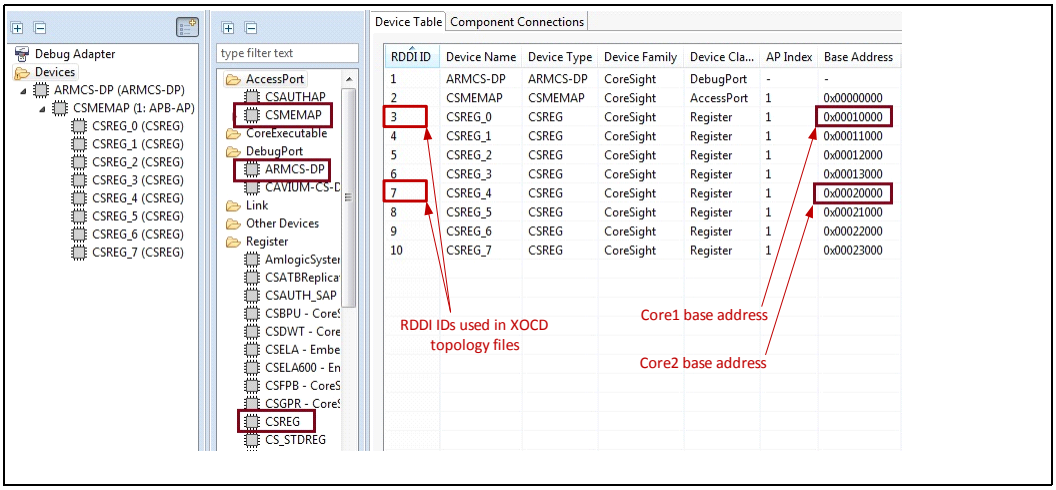


Figure 6–29. DS-5 Platform Configuration Editor Xtensa example.

Note: The TCP/IP address in this example must be cached to match the used setup.

By default, XOCD looks for an RV configuration file named `rv-conf.sdf` in the current directory, or if not found there, in the XOCD install directory. As indicated in Section 6.6.3, the `--rvconfig=<filename>` command-line option can be used to specify an alternate name and location of the RV configuration file.

6.11.2 XOCD Topology File for DSTREAM

Relative to the original (non-DSTREAM) topology file documented in Section 6.7, the following changes are necessary: In the `device` elements, the `xdm-id` attribute needs to specify the *RDDI IDs* of the CSREG templates used to access the given OCD module (see Section 6.11.1 for more details on the RV configuration file).

The XOCD installation files also include a topology file specifically configured for DSTREAM probes:

- `rv/topology-CSREGx2.xml` (two Xtensa cores with TRAX).installation.

The file has `xdm-id` that matches the example `rvc-CSREGx8.sdf` configuration file.

6.11.3 Debugging using the Serial Wire Debug Interface (SWD)

When using CoreSight, the Platform Configuration Wizard allows for select the Debug Adapter — JTAG or SWD. The selected interface is automatically handled by the DSTREAM probes and is hidden from XOCD. The probe takes responsibility of properly switching between JTAG and SWD, ensuring SWD transactions are properly executed in order to reach Xtensa Debug module registers in APB space.

6.11.4 Limitations

A given instance of XOCD can attach to only one DSTREAM probe.

6.12 Cadence Palladium VDEBUG

Cadence offers a Palladium Virtual Probe (VDEBUG), which can be instantiated into a hardware design to allow accessing Xtensa JTAG or Debug APB bus. The probe uses SystemVerilog DPI to receive debug transactions from a debugger. The VDEBUG deliverables are unrelated to XOCD—it's expected the deliverables will include sufficient information regarding instantiating the probe into a user project. The XOCD accesses VDEBUG using TCP/IP. Table 6–35 describes all attributes specific to the Cadence VDEBUG probe.

Note: XOCD is tested to work with Cadence VDEBUG version ‘37’, build ‘01.06.18’.

Table 6–36 describes all the topology file attributes specific to the Cadence VDEBUG probe.

Table 6–36. Cadence Palladium VDEBUG Probe Attributes

Attribute	Description
`bfm-clock`	VDEBUG virtual probe clock period in picoseconds. The probe uses this value as a clock reference.
`bfm-path`	VDEBUG virtual probe instance path. E.g. ‘`dut_top.jtagBFM`’.
`host`	Name of the Palladium host where the virtual probe is running.
`port`	Port number on the Palladium host where the virtual probe listens for commands.
`type`	Type of the virtual probe. Either ‘`jtag`’ or ‘`apb`’.

Table 6–36. Cadence Palladium VDEBUG Probe Attributes (continued)

Attribute	Description
`poll-low`	Number of clocks in the shortest "XOCD poll" to the virtual probe. Refer to Section 6.12.1 for details on XOCD poll.
`poll-high`	Number of clocks in the longest "XOCD poll" to the virtual probe. Refer to Section 6.12.1 for details on XOCD poll.
`jtck-div`	(JTAG ONLY). Set JTCK frequency by programming the clock divider. VDEBUG generates the JTCK clock by dividing the input clock frequency. The maximum JTCK is half of the VDEBUG input clock frequency and is achieved by setting this value to 1.

The Palladium VDEBUG virtual probe is described in the topology file as follows:

```
<controller id='Ctrl0' jtck-div='1' host='myhost'
bfm-path='dut_top.jtagBFM' bfm-clock='10000' poll-low='10'
poll-high='5000' module='pall_vprobe' type='jtag' port='8192' />
```

6.12.1 XOCD Poll

After the virtual probe is instantiated in a Palladium system, the emulator will move forward (consume time) only when the virtual probe simulator side executes a task that consumes time in the emulator. To move the emulator forward, it's important that the XOCD keeps executing a Polling command whenever it has no actual debug commands to send out. The polling is a special call from XOCD into the virtual probe and is called a poll tick. The poll tick size (number of emulator clock cycles to execute on each poll call) requires a special consideration from the user side, as the emulator is unresponsive while executing a poll tick.

A large poll tick makes the emulator execute many clock cycles without synchronizing with the simulator (a mechanism inherent to the Palladium). This allows for faster emulation speed; however, this results in increased debugger unresponsiveness. Smaller poll tick values will cause unnecessary simulator/emulator synchronizations whenever XOCD has no commands to send, which slows down the emulator speed.

The following polling scheme is adopted in XOCD's VDEBUG driver:

With each command from XOCD to the core, the current poll tick size gets set to `poll-low`, to make the virtual probe more responsive to the debugger. When there are no commands to the core, the polling value gradually increases to `poll-high`, enabling the emulator to run faster. The caveat of the scheme is the unwanted delay of the very first command that is issued after some idle—the command might be blocked from execution up to `poll-high` Palladium clock cycles.

IMPORTANT: Set `poll-low` and `poll-high` to the values that are most appropriate to your system.

6.13 XOCD Initialization Script

This section describes how to use XOCD with an initialization script. The primary purpose of script execution is to deal with systems with multiple scan chains, where selection of a desired scan chain must occur before you can access the target.

The option -I <filename> executes JTAG commands from script <filename> on XOCD startup, before any other JTAG command is executed.

6.13.1 Initialization Script Format

Script files are processed sequentially line by line, executing the specified series of IR scans, DR scans, TAP reset, and/or system reset.

Each line starts with a command or comment character in the first column, followed in some cases by space-separated length and data parameters. Note that the length is in decimal, and data bytes in hex. The maximum allowed line length is 2048 characters (spaces included). Empty lines are not allowed. Possible lines are as follows:

- `DR <length> <data>` Executes DR scan with specified data
- `IR <length> <data>` Executes IR scan with specified data
- `TRST` Executes TAP reset
- `SRST` Executes System reset
- `# <comment>` Comment line

where:

- `<length>`

  This is the total length in bits of the IR or DR data to scan over the entire scan chain, which is expressed in decimal format. For example, for a scan chain containing a single Xtensa processor's TAP, the length is 5.

- `<data>`

  This is the data to shift out onto the scan chain; that is, to shift into each TAP component during the TAP shift phase. The data must be formatted as a sequence of bytes in hexadecimal with a space(s) between each hex byte. The last (right-most) byte is shifted out first (lsbit first) into the TDI of the first TAP in the chain. If the length is not a multiple of 8 bits (not a whole number of bytes), the first byte is zero padded in its upper bits.

6.13.2 Initialization Script Example

The following example issues four JTAG commands:

```
TRST
IR   5    1C
DR   8    8A
DR   30   2B CD 12 34
```

1. The first command is a TAP reset.
2. The second command executes JTAG IR command 28 (hex 1C) of length 5. The resulting 5-bit sequence "11100" is sent least significant bit first, and the corresponding number of bits scanned in are displayed. For example, in a scan chain containing one Xtensa core, this command selects the 8-bit NAR, which alternates with the 32-bit NDR in consecutive DR scans.
3. The third command scans the DR register (normally selected by the TAP instruction sent in the last IR scan). In this case the DR is the 8-bit NAR, written with 0x8A (or binary: 10001010), which corresponds to a request to read DDR on the subsequent NDR scan.
4. The fourth command scans the DR register, which in this case is the 32-bit NDR. Normally, the selected DR does not change for a given IR, however, the NAR/NDR pair is special; they alternate when doing successive DR scans. Even though the Xtensa NDR is 32 bits long, here we used 30 bits for illustration of an odd multiple of bits. So here, 30-bit of DR (now mapped to NDR) are scanned: it is written, or scanned out, with 0x2BCD1234 (or binary: 10101111001101000100100011010 0), and the corresponding number of bits scanned in are displayed.

XOCD generates output similar to the following as a result of executing this script (where *BB* represents scanned-in data):

```
Execute JTAG script
 TAP Reset
 Scan IR (len:  5) 18 > JTAG > BB
 Scan DR (len:  8) 8A > JTAG > BB
 Scan DR (len: 30) 2b-cd-12-34 > JTAG > BB-BB-BB-BB
```

6.14 XOCD JTAG Output File

This section describes how to enable XOCD to record the executed JTAG commands.

The option `--scans=<filename>` records all the executed JTAG commands into the file `<filename>`.

6.14.1 Output File Format

JTAG record file is added a line each time an IR scans, DR scans, TAP reset, and/or system reset are executed by XOCD.

Each line starts with a command in the first column, followed in some cases by space-separated length and TDI/TDO bytes. Note that the length is in decimal, and TDI/TDO bytes in hex. Empty lines are not allowed.

Possible lines are:

- `DR <length> <TDI> <TDO>` DR scan with data scanned out and in
- `IR <length> <TDI> <TDO>` IR scan with data scanned out and in
- `TRST` TAP reset
- SRST System reset

where:

- `<length>`

  This is the total length in bits of the IR or DR data scanned over the scan chain, which is expressed in decimal format.

- `<TDI>`

  This is data shifted out onto the scan chain;, that is, shifted into each TAP component during the TAP shift phase. The data is formatted as a sequence of bytes in hexadecimal with a space(s) between each hex byte.The last (right-most) byte is shifted out first (lsbit first) into the TDI of the first TAP in the chain. If the length is not a multiple of 8 bits (not a whole number of bytes), the first byte is zero padded in its upper bits.

- `<TDO>`

  This is data shifted in from the scan chain, that is, shifted from each TAP component during the TAP shift phase. The data is formatted as a sequence of bytes in hexadecimal with a space(s) between each hex byte. The last (right-most) byte is shifted in first (lsbit first) FROM the TDO of the last TAP in the chain. If the length is not a multiple of 8 bits (not a whole number of bytes), the first byte is zero padded in its upper bits.

6.14.2 Output File Example

The output file format closely resembles the initialization script format, with the addition of scanned-in data appended at the end of each line. In fact, the output can sometimes be used as an initialization script (assuming the required sequence is not input dependent).

Borrowing from the example in Section 6.13.2, the following example shows four recorded JTAG commands:

```
TRST
IR   5    1C   01
DR   8    8A   88
DR   30   2B CD 12 34    11 22 33 44
```

1. The first command is a TAP reset.
2. The second command executed is JTAG IR command 28 (hex 1C) of length 5. The resulting 5-bit sequence "11100" was sent least significant bit first. For example, in a scan chain containing one Xtensa core, this command selected the 8-bit NAR. The last shown byte, 01, is the data received from the TDO pin during the scan. In this case, it's a 5 bit sequence "00001".
3. The third command scanned the DR register (normally selected by the TAP instruction sent in the last IR scan). In this case the DR is the 8-bit NAR, written with 0x8A (or binary: 10001010), which corresponds to a request to read DDR on the subsequent NDR scan. The last byte is the value scanned-in from the TDO pin during this DR scan.
4. The fourth command scanned the DR register, which in this case is the 32-bit NDR. Normally, the selected DR does not change for a given IR, however, the NAR/NDR pair is special; they alternate when doing successive DR scans. Even though the Xtensa NDR is 32 bits long, here we used 30 bits for illustration of an odd multiple of bits. So here, 30-bit of DR (now mapped to NDR) was scanned: it was written, or scanned out, with 0x2BCD1234 (or binary: 101011110011010001001000110100). The last four bytes shows the value scanned-in from the TDO pin during the execution of this DR scan.

6.15 Using XOCD to Debug the Debug Vector

Note: This section is not applicable to the cores implementing XEA3 exception architecture.

You can set up XOCD to allow debugging of a debug vector, such as XMON (see Chapter 19). If you are debugging the XMON code during a time that XMON is not performing any debug activity, no special handling is needed. This is because XOCD has control of the program execution; thus all the debug interrupts and exceptions will halt the core effectively, allowing the XOCD to take the control. However, if you need to debug XMON at the same time it is doing another task (such as debugging an application), XOCD must be able to hand off control to XMON for all the debug events that originate from XMON.

In the current implementation, support is only provided for handing off control for software breakpoints. That is, if XOCD does not recognize a breakpoint (it's not planted by XOCD), or the ICOUNT debug exception happens when XOCD is not executing a single step, it resumes execution at the debug vector address. This way, XMON is able to continue with its own debugging. You can select whether coded breakpoints are to be handled by XMON or XOCD. Note that handing off control for any other debug event is not currently supported.

There are three monitor commands that control the hand-off set up: enabling XOCD's support for breakpoints, enabling the hand off to XMON, and enabling or disabling XMON's support for coded breakpoints. See Section 6.19 for details.

6.16 Synchronous Debugging

The *synchronous debugging* feature allows synchronously stopping and restarting a group of processors using hardware support described in Section 5.8. A processor hitting a debug exception stops all processors in that group nearly simultaneously. For synchronous restart, XOCD sends instructions to restart all processors simultaneously. **Note:** When resuming a processor stopped as result of a Debug Stall input being asserted (see Section 5.9), XOCD does not explicitly send any resume instructions—the resume action is a part of the debug stall hardware operation.

There are two parts to enabling the software for this feature. First, the topology file must properly describe how processors are interconnected, so that the Xtensa OCD Daemon can know which groups of processors can be made to start and stop simultaneously. Second, synchronous debugging for any subset of processors in these groups is controlled by a set of individual debugging sessions, one per processor being debugged. These two parts are described in the following subsections.

6.16.1 Describing the Hardware Topology for Synchronous Debugging

The topology of the BreakIn and BreakOut interface connections between cores must also be reflected in the configuration file for XOCD. The `<target>` elements in the `application` section must include the additional attribute `sync-group`. It must be set to the same value across all processors that belong to the same group.

The following example describes a two-processor system that is wired as described above to provide synchronous debugging.

```
<application id='GDB0' module='gdbstub'>
  <target device='Xtensa0' sync-group='1' loose-sync='1'/>
  <target device='Xtensa1' sync-group='1' loose-sync='1'/>
</application>
```

Note: The topology of the DebugStall interface connections between cores does not need to be reflected in the topology file. This is because the debug stall is only a hardware feature that helps keeping cores better synchronized while debugging only a subset of them. From an XOCD perspective, a core in debug stall is not a core under debug so that it needs to be handled in any way. For example, a core entering debug stall, or a core leaving debug stall will be handled by only one master core, so no debugger (other then the one attached to the master core) needs to be involved. Nevertheless, a debugger (e.g. Xtensa Xplorer) can arbitrarily decide whether to stop all the stalled cores when a master core (under debug) stops.

6.16.2 Using the Synchronous Debugging Feature

With the OCD daemon running with a configuration file that describes the multiprocessor target's synchronous debug interconnect topology, synchronous debugging can begin.

Synchronous debugging can be controlled by either GDB directly (`xt-gdb`), or using the debugger from within Xtensa Xplorer. In GDB, synchronous debugging of the attached processor is turned on and off using the "`monitor sync on|off`" command. Within Xtensa Xplorer, synchronous debugging is controlled by an option on the pull-down menu in the Debug window.

By default, synchronous debugging is turned off for all processors. It is always off for processors to which no debugger is attached, and initially off when a debugger attaches to a processor. You can turn synchronous debugging on and off for each individual processor using multiple concurrent debugger sessions (one per processor).

For synchronous debugging to work, it must be turned on for at least two processors in the same sync group. To understand this better, let's first introduce some terminology. Within a given sync group (as defined in the topology file in Section 6.16.1), the subset of processors that have synchronous debugging turned on is called a *synchronous subset*. Once a synchronous subset consists of at least two processors, it becomes active. When active, processors within the synchronous subset proceed synchronously. That is, they start and stop together, from the point of view of the Xtensa OCD daemon.

Processors start as synchronously as the speed at which the JTAG clock propagates. When processors stop synchronously, the exact point at which each processor stops can differ by a few cycles or instructions. Factors include delays in the breakin/breakout network and what each processor was doing at the time. This effect is more visible when single-stepping a processor, which involves a very small number of cycles: other processors advance by an indeterminate number of instructions, possibly zero.[3]

3. This is also true when single-stepping *multiple* processors synchronously using `xt-gdb`: it is possible for some processors to not advance.

Note that the synchronous start of different cores on the same JTAG clock cycle can be achieved only if the synchronized cores share a similar resume mechanism. However, this is not always the case. When cores cannot be resumed in such a way (for example, when mixing RE and RD cores) setting `loose-sync='1'` will result in the serialized resume of the given core in respect to the others in the same group. In this case, the synchronized cores implement so-called “loose” synchronization. It is your responsibility to set this parameter for each core according to their system.

Note: The synchronous debugging support in the Xtensa OCD Daemon assumes that the only source of external debug interrupts is the breakin/breakout network. If another debug interrupt source causes the synchronous subset of processors to stop, the debuggers will not be notified, and thus will not indicate that processors have stopped. Thus, additional debug interrupt sources are not currently supported.

6.16.3 Synchronous Debugging Using Xtensa Xplorer

Xtensa Xplorer helps manage multiple synchronous debugging sessions. Within the synchronous subset, it ensures debugging sessions are either all running or all stopped.

Xtensa Xplorer only works with a single synchronous group at a time.

Refer to the Xtensa Xplorer Help topic, Synchronized debugging, for details about setting up a synchronized debug session.

6.16.4 Synchronous Debugging Using Xtensa Tools GDB (`xt-gdb`)

Synchronous debugging is also possible using command-line GDB (`xt-gdb`). You invoke a separate `xt-gdb` session for each processor being debugged.

The Xtensa OCD Daemon coordinates individual debugger requests so that processors act synchronously. The Xtensa OCD Daemon cannot cause a processor to resume execution when its debugger has control of the processor. Thus it must wait for all debugger sessions to tell their processors to resume execution before letting the whole subset resume execution synchronously. When a debugger has told processor A to run, but the Xtensa OCD Daemon is waiting for other debuggers to do the same before letting A run, processor A is said to be on *standby*. Once all processors in the synchronous subset are on standby, execution resumes synchronously.

Similarly, when the synchronous subset stops, only the debugger(s) whose processor caused the subset to stop[4] gets control back. Other processors remain on standby. This avoids the need to ask every debugger session to resume execution every time the subset is to resume execution.

You can always gain control of a processor on standby using `Ctrl-C`.

A few other cases are worth noting:

- When turning *off* synchronous debugging for a processor, the remaining processors resume execution if they are all on standby.
- If turning *on* synchronous debugging for a processor (which can only be done while the processor is stopped) causes it to join an existing subset that is already running, the existing subset automatically stops and enters standby mode.

6.16.5 Synchronous Debugging in the CoreSight Environment

When resuming devices attached to the DAP APB-AP port, synchronous resume is executed differently than the resume for the devices chained on the JTAG. With APB-AP, devices need to be resumed one after another. The final achieved synchronization is very loose, approximately within hundreds of JTAG clocks cycles. To achieve better synchronization, we recommend using the debug stall network (see Section 5.9).

6.17 Connecting to the Target in Power Shut-Off

The regular XOCD operation is to allow core and memory subsystems to power down when core execution is resumed from the debug interrupt. Also, the debug power domain is allowed to power down when XOCD is disconnected from the target. Similarly, all three power domains are forced ON when the core is debugged.

Note: XOCD executes power related operations only if the target includes a power shut-off mechanism (PSO). You can also achieve a certain level of control over this mechanism using GDB monitor commands (see Section 6.19).

A situation that needs further consideration is the transition between the core being released (not under the debugger's control—freely running) and the stopped core (after debugger issues command to stop the core):

4. All processors stop through a debug exception. If the cause is an external debug interrupt, assumed to come from the breakin/breakout network, the processor remains on standby. Otherwise control returns to the debugger. That way, generally only one debugger gets control, although it is possible for multiple debuggers to get control if the timing is just right.

When XOCD receives a request to stop the core, it issues the debug interrupt, followed by waking up the core so the interrupt can be taken. This order of events allows for stopping the core right after the wake up (if the core is currently in PSO) before it executes any instruction. However, waking up the core might not be desired in debugging scenarios where you just want to observe the system state and then resume the core from the point where it was stopped. If the core is in PSO, that means the debugger needs to be able to leave the core in PSO after examining the state. However, this is not possible as the core will simply resume the execution from the reset vector, resulting in the system state perturbation. To cope with this issue, XOCD allows for controlling the wake up after the debug interrupt using GDB monitor commands. If the wake up is not desired upon the debug interrupt, XOCD won't be able to gain control over the core in PSO. In this case, XOCD continues issuing debug interrupts until it succeeds (for example, the core powers up on a request from another core). In parallel, the Debug module is still accessible for the debugger to issue the wake up request (see Section 6.19). XOCD stops looping the debug interrupt request after it sees the debug interrupt is taken.

6.18 Core and System Reset

Upon receiving the GDB reset command, XOCD will normally reset the target core through the PWRCTL register. XOCD then asserts the debug interrupt while the core is in reset, effectively stopping the core right after the core reset signal is removed, before the core executes any instruction.

The behavior of the GDB reset command can be changed to reset the whole System instead of a single core (see the `monitor default-reset` command in Section 6.19). The term "System" applies to any hardware logic, which is reset using the RESET (SRST) pin of the JTAG connector (see Section 2.3.10). In addition, you can explicitly select to reset the debug and JTAG logic during the system reset using the topology file options (see `reset` option in Section 6.7.4). Finally, System reset will always reset all the cores currently attached to the GDB, using the PWRCTL register.

Be aware that the debug logic reset also resets the TRAX and Performance Counter logic, thus any measured or collected data is lost. Therefore, the debug reset is not to be used if the intent of the current debug session is to obtain previously collected data.

Issuing debug reset also has an effect on XOCD interruption of the core after the core reset. Since the XOCD is unable to assert the debug interrupt during the core reset (so it's taken after the core reset is released) the core is now freely running after the reset. XOCD will eventually stop the core but the fact that core was running for some time might have different undesired side effects.

6.19 Supported GDB `monitor` Commands

The GDB `monitor` command relays an arbitrary command string to its debug agent, such as the Xtensa OCD Daemon. In other words, when debugging over OCD, all arguments to the `monitor` command are sent as a string to the Xtensa OCD Daemon. Thus, the set of supported parameters or subcommands is a function of the debug agent (the Xtensa OCD Daemon in this case), not of GDB. This section describes the `monitor` commands that the Xtensa OCD Daemon understands.

Note: The default setting of a given parameter (`on` or `off`) is indicated by *(default)*.

- `monitor`

  Displays summary usage information.

- `monitor exe <hex_opcode_bytes>`

  Executes a single processor instruction, provided as a stream of bytes in hex. This is a backdoor mechanism sometimes used in elaborate GDB macros, such as for dumping the state of caches or MMU TLBs. Certain classes of instructions must be avoided (for example, instructions that stall, take an exception, or affect the state used by the debugger; there may be others); the result of attempting to execute them is undefined.

- `monitor override [on|off]`

  Allows interruption of the core during operations that are normally not interruptible by the debug interrupt, even if it means that the state of the system becomes corrupted. The default state is off.

- `monitor sync{-stall|-break} [on|off]`

  Run this core synchronously with others whose GDB has executed this command. The default is `off`. Run this core independently (see Section 6.16.2).

  The synchronization is achieved either using the stall network (`sync-stall on`) or using the break network (`sync-break on` or `sync on`).

  Note: `monitor sync-stall off` will disable the RunStall input (clear `DCR.RunStallIn` bit as discussed in Section 5.5.2) only if not enabled by the `monitor force-runstall on` command.

- `monitor force-runstall [on|off]`

  Enables the Runstall input (set `DCR.RunStallIn` bit - see Section 5.5.2). Also, disables `XOCDMode`—the stall output signal (`DCR.DebugModeOutEn` cleared) only if not enabled by the `monitor sync-stall on` command.

- `monitor loglevel <N>`

  Sets XOCD logging verbosity level to <*N*> (in decimal) ranging from 0 (none) to 99 (most detailed). XOCD logging must already be enabled with the `-d` option. See Section 6.6.3 for details.

- `monitor tracelevel <N>`

  Turns on tracing of internal function calls, and sets the detail level to <*N*> (in decimal) ranging from 0 (none) to 99 (most detailed). See also options `-T` and `-L` in Section 6.6.3.

- `monitor tracefile <filename>`

  Directs tracing of internal function calls to the specified file. The path is relative to the Xtensa OCD Daemon current directory. See also options `-T` and `-L` in Section 6.6.3.

- `monitor verify-read [on|off]`

  Enables (`on`) or disables (`off` - default) checking for completion of memory load instructions. It is safe to disable only if loads are known to always have low latency relative to JTAG probe speed.

- `monitor verify-write [on|off]`

  Enables (`on`) or disables (`off` - default) checking for completion of memory store instructions. It may be unsafe if stores have *both* long latency and low throughput relative to JTAG probe speed (usually unlikely). Note that this greatly reduces download speed.

- `monitor verify-all [on|off]`
- Enables (`on`) or disables (`off` - default) checking for completion of ALL instructions executed via JTAG. Note that this can significantly slow down memory accesses and other debugger operations. Some instructions are verified even when this option is `off`.

- `monitor scanio <irvalue> <drwidth> <drvalue>`

  Executes a JTAG scan to the current core's TAP controller (with all other TAPs on the chain in bypass).

  - <*irvalue*> is the Xtensa TAP instruction register value to issue (5 bits wide)
  - <*drwidth*> is the width in bits of the TAP data register selected by that instruction register (up to 64)

- <*drvalue*> is the data register value shifted to the TAP (accepted by the TAP in CaptureDR state).

This command displays the data register value shifted back from the TAP (driven by the TAP in UpdateDR state).

- `monitor xdmreg-rd` *`REG`*

  Reads the debug module register *REG*.

- `monitor xdmreg-wr` *`REG VAL`*

  Writes the debug module register *REG* with value *VAL*.

- `monitor breakpoint [on|off]`

  Allows XOCD to handle breakpoints (the default is off).

- `monitor default-reset [core|system]`

  Selects the hardware logic to be reset upon gdb '`reset`' command:

  - `core` - This default command resets the core through the PWRCTL Debug register.
  - `system` - Issues reset on a $\overline{\text{RESET}}$ JTAG connector pin (see Section 2.3.10). Also resets all the cores in the system that are currently connected to GDB. Finally, it resets JTAG and debug logic, if specified in the topology file (See `reset` option in Section 6.7.4).

- `monitor reset [jtag|debug]`

  Resets JTAG or debug logic. This option is not to be used during regular debugging. It is needed in cases when the corresponding hardware logic is in an unknown (bad) state and needs to be reset. Upon executing the debug reset, the core is still under the debugger's control.

- `monitor {i|d}cache invl|init|wb-invl]`

  Performs an operation on either the data cache (`monitor dcache...`) or the instruction cache (`monitor icache...`). The operation is either to invalidate, initialize or writeback dirty data followed by cache invalidate, respectively. For instruction cache, the last operation performs the invalidation only.

- `monitor poll [on|off]`

  When this debug option is off, a periodical check for the target status is not performed. The running core is assumed as running further on and vice versa for the stopped core. This command allows for disconnecting the probe from JTAG when GDB is idle.

- `monitor int-wakeup [on|off]`

  When this command is on (the default setting), a GDB request to interrupt the core will also wake up the core, if the core is in PSO. When off, XOCD only issues the debug interrupt that cannot be taken by the core at that point.

- `monitor failure-mode [on|delayed|off]`

  When on, the debugger will be prevented from performing any memory accesses. Use this in cases when the next access to a memory subsystem is likely to hang because the previous access didn't complete. This setting still allows the debugger to access the processor's registers in order to debug the memory subsystem issues, without the possibility that the debugger will issue a memory access that could hang the processor in the middle of a debug operation.

  When `delayed`, the failure mode setting changes automatically to `enabled` just before the next time that XOCD resumes execution of the target processor. Use the `delayed` setting when you expect the memory subsystem to hang after the next resume (because you expect that the program running on the processor will cause a hang in the memory system). Enabling the failure mode in such a way ensures that the next time that the debugger regains control, it will be prevented from accessing memory.

- `monitor interrupt-all-conditions [on|off]`

  Certain memory operations cannot be interrupted with a debug interrupt. This option allows for interrupting the core even if the memory subsystem is holding the core (when on).

- `monitor debug-xmon [on|off]`

  Hands off the handling of debug exceptions to XMON, for exceptions not recognized by XOCD, such as software breakpoint not planted by XOCD, coded breakpoints, and instruction stepping exceptions while XOCD not actually stepping.

- `monitor xmon-coded-bp [on|off]`

  Makes XMON handle coded breakpoints, instead of XOCD.

- `monitor ctrl-dmi-support [on|off]`
  The controller level of XOCD can implement a Direct Memory Interface (DMI), which allows XOCD to simply send all GDB's memory accessing instructions to the controller and the probe to access the system memory using the probe's existing capabilities. This command enables/disables DMI support. Note that accessing the memory directly might leave the processor's caches in an incoherent state. Also, it is the user's responsibility to account for the processor's memory translation settings as XOCD passes-through the virtual memory addresses, as received by GDB.

7. OCD Debugger Software Tools Development

This chapter describes implementation details and issues relevant to writing debuggers that use OCD. It is intended for developers of OCD-based debugging tools for Xtensa processors.

Note: The architecture of the OCD and TAP controllers has changed subtly from processor release to processor release. Designers wishing to support previous versions of the Xtensa processor should refer to the documentation for that processor version.

Xtensa LX processors support Xtensa Exception Architecture2 (XEA2); Xtensa NX processors support Xtensa Exception Architecture3 (XEA3) only. With the introduction of XEA3, the Xtensa Debug architecture is significantly changed. The changes affect the concepts and explanations discussed in this chapter, so some sections are marked as being applicable to either XEA2 (and earlier) or XEA3.

7.1 Debugging Methods Terminology

Various approaches can be used to debug Xtensa code. Determining which one is preferable or applicable depends on context, for example, what type of code is being debugged and in what environment. For the purpose of this discussion, a debugging method is some set of mechanisms and conventions that allow one to:

- Start and stop Xtensa code.
- Set and remove breakpoints (perhaps various kinds).
- Gain control when the code hits a breakpoint, single-step the code, and access relevant core state and memory state.

Five debugging methods are presented here, out of three basic debug mechanisms:

1. **OCD (On-Chip Debug)**
   OCD is enabled, so that Debug exceptions stop the processor. The host debugger has control through the Debug module. The processor is normally in *Running* state (see Section 5.2) and enters the *Stopped* state upon debug exceptions. The host uses the *Stepping* state to feed instructions to the core in order to control the core and access its state and memory.
2. **Debug Vectored (XEA2 and earlier)**
   OCD is disabled, or not configured, so that Debug exceptions are handled by the core, which branches to the debug exception vector. The core is always in *Running* state (see Section 5.2). This allows the core to always service interrupts of higher priority than DEBUGLEVEL while debugging, and to do task level debugging such

that other threads continue to run while one is being debugged, so is sometimes referred to as a *realtime debug mode*.
Three variants of this approach are:

a. **Classic Debug Stub**
The debug exception handler communicates with the host debugger using a custom communication channel, *e.g.*, serial line, Ethernet, etc.

b. **DDR-Based Debug Stub**
The debug exception handler communicates with the host debugger using the DDR register, across the Debug module's Access Port. Note that this approach uses registers normally used with OCD, but with OCD disabled.

c. **Local**
The debug exception handler communicates directly with a debugger running locally on the core. An example is Linux debugging applications natively.

3. **General Exception Vectored**
This approach avoids using any Debug option features: Debug exceptions are not used, nor are instruction and data address breakpoints (IBREAK and DBREAK*n*), ICOUNT, nor the standard BREAK or BREAK.N instructions. This allows this method to be used independently of the others above: for example, it may be used for task level debugging, while simultaneously allowing OCD debugging of the whole processor. Also, the processor debug option need not be configured.
Breakpoints are implemented using an illegal or other instruction that causes a general exception (user vector, kernel vector, or double exception). Lacking ICOUNT, single-stepping is done by temporarily planting a breakpoint at all possible next instructions, determined by decoding the instruction to be single-stepped and looking at relevant registers. The next instruction is normally the fall-through PC, the LBEG register value when at the last instruction of an active zero-overhead loop, or the branch target for unconditional branches (jumps and calls). Conditional branch instructions have two possible next instructions (unless the instruction is decoded enough to resolve which one will be taken).
Exception and interrupt vectors are typically not debugged using this approach.
This approach may not always be practical given the complexity of decoding the instruction to be single-stepped. This is complicated in processors configured with FLIX where the potential encodings are many and varied.

The Xtensa OCD Daemon supports OCD debugging at the processor level. Historically, RedBoot and XMON have been processor level Classic Debug Stubs, and used a mix of Classic Debug Stub and General Exception Vectored approaches at the task level.

Or, a single debugging method might support both breakpoints using the BREAK instruction and breakpoints that cause general exceptions (for example, an illegal instruction). The four methods are not an exhaustive list; they are just a representative set for the purpose of describing Xtensa's debugging support.

Further sections within this document will refer to the above debugging methods terminology.

7.1.1 XEA3 Monitor Debug Mode

In XEA3, Monitor Debug mode is introduced and is a processor's configurable option. The Monitor Debug mode is the replacement for the Realtime/Vectored debug mode introduced in Section 7.1, and its focus is to remove much of the debug-related specialty that was affecting interrupts and exceptions. Essentially, the notion of the debugging interrupt level is removed so the debug interrupts are handled just like any other interrupts. In addition, the debug exceptions are no longer treated as debug interrupts but as regular exceptions.

7.2 Considerations when Executing Xtensa Instructions via OCD

This section expands on the mechanism for executing Xtensa processor instructions via OCD described in Section 5.2.2 "Executing Xtensa Processor Instructions".

7.2.1 Example Sequences for Executing Xtensa Instructions

A typical sequence is the following:

1. Enabled the OCD and issue a Debug Interrupt (see Section 5.2.2); make sure DSR was read so that it is now clear. This is performed by setting the EnableOCD bit in DCR OCD register and setting the DebugInterrupt bit.
2. Write the Xtensa instruction opcode to the DIR0EXEC register, which will also initiate the instruction execution.

   Loop:
3. Read the DSR register.
4. If either ExecException or ExecDone bit is set, end the loop; instruction has completed. Else loop to wait for completion

In case when the instruction operates with the DDR register, step 2 from above can be either of the following:

a) Write the data to the DDR register and the execute step 2.

b) Write instruction to DIR0 register and then write the data to DDREXEC register.

The remaining subsections provide additional information and highlight relevant issues one should take into account when executing Xtensa instructions using this mechanism.

7.2.2 Cache Issues

Executing instructions from DIR does not affect the instruction cache (if present), unless it is an instruction cache control instruction (IHI, *etc*.). However, any load or store instruction executed accesses and affects the data cache (if present and enabled) as any normal load or store would. For example, on loads to memory set as cacheable, misses update the data cache and hits read from the data cache rather than memory.

Stores on systems with write-back caches also change the state of the cache. In the case of writing out new processor instructions, it might also be required to execute a data cache write-back instruction (DHWB, *etc*.) to flush the data cache to memory. Typically, a debugger writes a block of data with each data (a memory word) at successive locations in the memory. Thus, DHWB is needed only when writing at the end of the data cache line.

Also, the DHWBI instruction might be needed to cast-out a dirty cache line before writing directly to RAM. Typically, a debugger issues DHWBI when starting with the new cache line.

If writing to the start of instruction cache line, IHI needs to be executed in case the host debugger is downloading the program code.

Note: Processors that have the MMU option have additional issues surrounding caching. Readers interested in producing host software using OCD for Xtensa processors for MMU should keep in mind that the OCD mechanism is an instruction insertion mechanism and that instructions executed with this mechanism have no special privileges. As a consequence, all memory references occur within the context of the current ring. Because the processor's *Stopped* state is entered through a debug exception, these instructions begin execution at ring level 0. The *Xtensa Instruction Set Architecture (ISA) Reference Manual* contains the details of the processor's behavior when the MMU option is selected.

7.2.3 Saving Core State during Debugging

On detecting entry to OCD, normally the external debugger must save all core state that will or might get clobbered by the debugger operations. Normally the debugger will save ARs used by the debugger, PC, PS and other common special registers. In addition, the debugger saves the registers that are present in the particular configuration, which depends on processor configuration options and the processor's release.

7.2.3.1 Saved Exception State in XEA2 or Earlier

A Debug Interrupt saves the current PC into the EPC corresponding to DEBUGLEVEL. In the *Stopped* and *Stepping* states, the exception model is the same as in the *Running* state. All interrupts, regardless of their level, are ignored. Exceptions, on the other hand, are not maskable and will be taken while executing instructions in the *Stepping* state. This will change the exception state of the processor. Consequently, if the host software can cause the processor to take an exception, then it must save the exception state on entering the *Stopped* state and restore the state back before entering the *Running* state.

Note: With XEA1 (T1050 and earlier), a side-effect of allowing exceptions to be taken during the *Stepping* state is that it lowers the current interrupt level of the processor. All processor states (EPS, EPC, PC, *etc*.) are updated. However, the processor continues to be in the *Stopped* and *Stepping* states and does not execute instructions from the cache or memory.

7.2.3.2 Saved Exception State in XEA3

Note that in XEA3, PC and PS are accessed directly (see Section 7.3.5.4), as the debug interrupt no longer has the exception semantics. It is expected that the debugger will also save the imprecise exception state (see Section 7.2.5), but only after it has waited for all imprecise exceptions to complete. For this, the EXCW instruction can be used.

Instructions that are used to save all such state are not expected to raise an exception. However, before doing anything that might cause an exception (for example, load or store) the debugger should set MS and/or PS for desired level of privilege and should clear imprecise exception state.

Note that imprecise exceptions are masked while in OCD mode. Assuming the debugger wants to detect any imprecise exceptions (for example, after loads/stores) it must read relevant imprecise exception state to determine which ones are pending (see Section 7.2.5).

7.2.4 Imprecise Exceptions

Some exceptions by their nature do not appear quickly enough for the processor to stop on the instruction causing the exception. Examples include uncorrectable ECC errors, floating point exceptions, gather/scatter exceptions, write bus errors that occur after the instruction commit point, and errors that occur on cache cast-outs.

Imprecise exceptions have some state associated with them that must be checked to see if an OCD-executed instruction generates one such exception.

The Xtensa instruction set currently specifies MESR, IEVEC, and IEEXTERN registers that collect information on various imprecise exceptions. These registers must be saved prior to debugging.

The debugger might also want to detect any imprecise exceptions (e.g. after loads/stores). To do so, it must read relevant imprecise exception state to determine which ones are pending, as described above. Note that it may be more efficient to perform this check after some block of memory has been accessed rather than after each load or store.Dealing With Interrupts

While the processor is in the *Stopped* and *Stepping* states, interrupts are not taken regardless of their level. This includes NMI. On returning from Debug Exception, if there are any pending interrupts, they will be taken right away. NMI interrupts that occur while in the *Stopped* and *Stepping* states are ignored.

7.2.5 Timer Freeze

Xtensa processors stop CCOUNT when stopped. Note that because of pipeline and related delays when stopping and resuming processor execution, the difference in CCOUNT between two successive instructions will be markedly higher if a debug interrupt is taken (in *Stopped* state) and then resumed between the two *Running* state instructions, than if the debug interrupt had not occurred.

7.2.6 Exceptions Occurring During Stepping State

Exceptions are reported in the DSR register with the DSR.ExecException bit. PS.EXCM is set to 1, PS.INTLEVEL is unchanged.

Note: In XEA1, PS.EXCM does not exist and PS.INTLEVEL is set to one when an exception occurs. This means that instructions executed in the *Stepping* state that cause exceptions can cause the processor interrupt level to descend from the DEBUGLEVEL to interrupt level one.

7.2.7 Debugging a Debug-Preemptive Interrupt Handler

A debug-preemptive interrupt handler cannot itself be debugged with a processor using *Vectored Debugging* (in *Running* state), at least not with the usual techniques of setting breakpoints in the handler and/or single-stepping through it. That would require stopping the handler, which contradicts its purpose. Other debugging techniques must be used (for example, simulation), and/or the debug-preemptive interrupt handler should be sufficiently simple as to require little or no debugging. Perhaps some possible damage can be tolerated or can be avoided in a special environment for the sole purpose of debugging the critical handler. If this is the case, OCD can be used, with some adaptations. Adaptations are required because debug exceptions never occur at PS.INTLEVEL ≥

DEBUGLEVEL. Possible adaptations can include using a lower-level interrupt or polling the critical interrupt via INTPENDING in software and handle it at a lower interrupt level. It is also possible to use a different mechanism than the break instruction for setting a breakpoint (for example, a call to a special routine that disables interrupts via INTENABLE, lowers the interrupt level, calls the break instruction then returns) and single-step with a lower PS.INTLEVEL, but with appropriate interrupts disabled by the debug host software (taking care to properly handle instructions that affect or read the current interrupt level).

7.2.8 OCD with Non-Critical Debug-Preemptive Interrupts

Note: This section frequently uses DEBUGLEVEL (the debug interrupt level). For XEA3, the following text applies assuming that DEBUGLEVEL is simply the chosen level of the debug interrupt.

If an Xtensa core is configured with debug-preemptive interrupts, and these interrupts do not need to be serviced while debugging, then it is possible to use OCD, with certain limitations and requirements.

First, if debug-preemptive interrupts are ever enabled and can occur, they must be disabled immediately after the processor stops, then later re-enabled prior to resuming to run the processor, to prevent further instructions executed in the *Stepping* state from being randomly killed by the debug-preemptive interrupts. The sequence to disable debug-preemptive interrupts is done in the *Stepping* state. Even though its instructions can be killed themselves, it can be made to work because the instructions do not otherwise cause exceptions and are not long latency. Each level of debug-preemptive interrupts can only be taken once (because they increase PS.INTLEVEL) in the disable sequence. This disable sequence usually consists of executing 'RSIL ar,15' until it completes without exception, then saving and clearing INTENABLE so that further instructions can change PS.INTLEVEL (*e.g.*, by causing a level-1 exception) without undesirable consequence. The re-enable sequence can consist of setting PS.INTLEVEL to 15 again, then restoring INTENABLE; then after possibly some other steps that must not affect PS.INTLEVEL, exiting the *Stopped* state which sets PS to EPS[DEBUGLEVEL].

Second, note that while debug-preemptive interrupt handlers are running, PS.INTLEVEL ≥ DEBUGLEVEL debug exceptions are neither taken nor latched. So, a Debug Interrupt request from the host debugger that happens to take effect while a debug-preemptive interrupt is running will be ignored. This means that debug-preemptive interrupt handlers should preferably exit quickly and not be active most of the time. This also implies that the host debugger software must loop issuing the Debug Interrupt until it actually takes effect, because it is not guaranteed to cause a debug exception. This issue actually exists for any code that sets PS.INTLEVEL at or above DEBUGLEVEL, independently of the presence of debug-preemptive interrupts. But when there are no debug-preemptive interrupts, it is usually possible to write code that always maintains PS.INTLEVEL below DEBUGLEVEL; with the presence and occurrence of such interrupts this is not possible.

7.3 Miscellaneous OCD Issues

This section covers debugging issues not covered elsewhere in this guide.

7.3.1 Ways to Lose Control of the Processor

There are a number of ways in which one can lose control of the processor so that a core reset is required. That is, no OCD sequence can regain the processor unless it is reset. These include (but are not necessarily limited to):

- Executing WAITI when in the *Stopped* state.
- Executing a load from a memory-mapped device that delays response on the bus forever.

7.3.2 Connecting to the Processor in an Unknown State

It may sometimes be desirable to connect a host debugger to a running system with minimal or no disruption. For example, one may want to have the host debugger gain control if any debug exception occurs while it is connected, or simply inquire as to the state of the processor DSR.Stopped indicates whether or not the processor is the *Stopped* state. This information allows the host debugger to know the appropriate action to take to gain control of the processor. Releases prior to T1040 did not have this bit in the DSR register.

7.3.3 How to Single-Step Xtensa Instructions

There are two ways a host can choose to single-step Xtensa instructions in a given program, when using OCD: running the instruction in either *Running* or the *Stepping* state.

7.3.3.1 Single-Stepping in Running State (XEA2 or Earlier)

In this method, the host debugger sets up ICOUNT and ICOUNTLEVEL using OCD and then returns to the *Running* state to perform the single-step.

Recall that ICOUNT increments for every instruction executed at CINTLEVEL < ICOUNTLEVEL.[5] An ICOUNT debug exception is taken at the beginning of an instruction if ICOUNT would increment to zero as a result of executing that instruction. If such a debug exception is taken, ICOUNT does not actually increment to zero, that is, it remains at 0xFFFFFFFF.

Hence to single-step <*n*> instructions, ICOUNT is set to –(<*n*> + 1). For example, to single-step a single instruction, ICOUNT is set to –2 (or 0xFFFFFFFE).

ICOUNTLEVEL is typically set to either 1 or DEBUGLEVEL, depending on what is being debugged. Setting ICOUNTLEVEL above DEBUGLEVEL is not useful for single-stepping, because debug exceptions such as the ICOUNT trap will not be taken at PS.INTLEVEL ≥ DEBUGLEVEL (such as when a debug exception handler is running). Setting ICOUNTLEVEL to DEBUGLEVEL allows single-stepping through exception and interrupt handlers (for interrupts at levels lower than DEBUGLEVEL). Setting ICOUNTLEVEL to 1 allows single-stepping through user code while skipping through most exception and interrupt processing—except where part of the interrupt or exception processing occurs in C and thus where CINTLEVEL drops back down to zero during that time. In general, ICOUNTLEVEL should be set to at least one greater than the CINTLEVEL of the code being debugged. This of course (as already mentioned) implies that using this mechanism, one cannot debug code running with PS.INTLEVEL at or greater than DEBUGLEVEL.

Note that it is very important to ensure PS.INTLEVEL is at DEBUGLEVEL (or higher) before setting up ICOUNT and ICOUNTLEVEL. In the old exception architecture, it is possible, for example, for PS.INTLEVEL to have dropped down to 1 if an exception occurred during the execution of a previous Xtensa instruction in the *Running* state. If that were the case, and PS.INTLEVEL was not restored to a proper value (*e.g.*, using RSIL or WSR to PS) prior to setting up ICOUNT and ICOUNTLEVEL, then ICOUNT may start incrementing for instructions executed in the *Stepping* state. This would result in the Xtensa instruction to be single-stepped not executing, because ICOUNT would already have incremented to 0xFFFFFFFF (or if multiple instructions were being stepped, fewer instructions than requested would end up executing).

5. CINTLEVEL is defined in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* as PS.INTLEVEL in XEA1 (the old exception architecture), and as max (PS.EXCM * EXCMLEVEL, PS.INTLEVEL) in XEA2 (the new exception architecture).

There is one caveat with the ICOUNT method of single-stepping. If one debugs exception handlers as well, (*e.g.*, with ICOUNTLEVEL=DEBUGLEVEL) when single-stepping an instruction, if an interrupt or exception is taken instead, one instruction of the exception vector will be executed (rather than just placing the PC at the start of the vector and returning control to the debugger before the first instruction at the vector).

Another point that must be considered is that certain instructions should not be single-stepped, but should be emulated by the host debugger instead, or the code should generate an error in some cases. This includes instructions that set PS.INTLEVEL ≥ DEBUGLEVEL, that access ICOUNT or ICOUNTLEVEL, RFI for levels at or above DEBUGLEVEL, and perhaps others.

7.3.3.2 Single-Stepping in Running State (XEA3)

XEA3 removes debug interrupt level and treats debug exceptions as regular exceptions. These architectural changes resulted in the removal of the ICOUNT and ICOUNTLEVEL mechanism for single-stepping, requiring a new architectural mechanism for single-stepping.

The OCD support for single-stepping is added through the StepRequest bit in the Debug Control register (see Section 5.5.2).

Monitor mode single-stepping is not yet supported in XEA3. As a substitute, the debug monitor code implements a scheme that could use software or hardware breakpoints after each instruction to closely mimic the single-stepping behavior.

It is also theoretically possible to single step by loading the next instruction that would be executed into the DIR register and executing it. However, this method of single-stepping becomes quite complex for the controlling software. Consider the case of the looping registers: when single-stepping through code that used the Xtensa zero-overhead loop, the host software must emulate the function of the looping registers because the looping features cease to function when the processor is in the *Stopped* or *Stepping* state. Consequently, it is not recommended that this type of single-stepping be used.

7.3.4 Determining Core Configuration Attributes Through the OCD

Generally, the configuration of a given Xtensa core is not detectable through the external debugger. It is best to have the host software informed of the processor configuration by some “out-of-band” process.

But determining some basic configuration parameters of a core to which one is connected is still useful, whether for sanity-checking against expected configuration parameters, or to gain enough information to access most of a core's state.

Determination of endianness can be detected by reading the OCDID register.

If endianness is known, many aspects of the core configuration can be validated by executing various Xtensa instructions whose results depend on core configuration, and testing for expected values. For example, the number of registers in the physical register file is easy to deduce. Set WINDOWBASE to 0, issue a '`rotw -1`' instruction, then read back WINDOWBASE; the resulting window base is the number of registers minus four, divided by four.

7.3.5 Accessing Core State and Memory

When implementing OCD software (see Section 7.1), the state of the processor and of memory must be accessed by executing Xtensa instructions in the *Stepping* state. These instructions must transfer the desired state to or from DDR, where the host debugger can read or write them through the OCD access ports.

7.3.5.1 Address Registers

The simplest case is accessing an address register in the current register window. For example, to read a5, one would execute:

```
WSR a5, DDR
```

and then read the value of a5 from DDR through the Debug module. The sequences to achieve this would be something like (assuming the JTAG is used):

1. <Reset the TAP if not in a known state>
2. Set the TAP instruction to select the TAP address register (NAR)
3. Write NAR to address the DIR0EXEC register
4. Write the TAP data register (NDR) with the encoding of the instruction 'WSR a5,DDR'
5. Write NAR to address the DSR register
6. Read NDR to obtain the value of DSR
7. If DSR.ExecDone is set, go to step 8
   otherwise, if DSR.ExecException is set, process error
   otherwise, go to step 5
8. Write NAR to address the DDR register
9. Read NDR this provides the value of a5

To access an address register outside the current window, one would first execute an appropriate '`ROTW <n>`' instruction to put the desired register within the current window, then use a sequence such as above (then eventually restore the WINDOWBASE, *e.g.*, using another '`ROTW <n>`' instruction).

7.3.5.2 Special Registers

Special registers are accessed through an address register. This requires saving the chosen address register, assuming it must be restored to its prior value. For example, to read PS using a3 as the intermediate address register, one would execute:

```
WSR a3, DDR
<read DDR through Debug Module to get value of a3>
RSR a3, PS
WSR a3, DDR
<read DDR through Debug Module to get value of PS>
<write DDR through Debug Module to restore value of a3>
RSR a3, DDR
```

Note that the last two accesses to DDR through the OCD can be done simultaneously, by shifting in the restored value of a3 in DDR while the PS value is shifted out. Of course, if a3 will only be restored later because multiple special registers will be accessed, then the DDR read and write steps can remain separate.

To write PS using a3, a similar sequence is used:

```
WSR a3, DDR
<read DDR through Debug Module to get value of a3>
<write DDR through Debug Module to set new value of PS>
RSR a3, DDR
WSR a3, PS
<write DDR through Debug Module to restore value of a3>
RSR a3, DDR
```

Again in this case, consecutive OCD read and write of DDR are present and can be combined in a single OCD data register scan operation.

7.3.5.3 PC and PS in XEA2 or Earlier

The PC and PS of the code being debugged are usually accessed in EPC[DEBUGLEVEL] and EPS[DEBUGLEVEL], rather than directly in the current PC and PS, which are not accessible as a special register.

7.3.5.4 PC and PS in XEA3

The PC and PS of the code being debugged are accessed as live registers. Note that PC can be read out using the CALL0 instruction.

7.3.5.5 Memory

Memory can be accessed using conventional load and store instructions. All the usual rules apply, for example in regards to alignment requirements, *etc*. Similarly to special registers, memory must be accessed through address registers, but usually through two of them: one to hold the memory address and another to hold the data to load or store.

Where a debugger does not know a priori whether the memory it is writing is code or data, it must invalidate the corresponding addresses in the instruction cache (if any) to avoid a stale instruction cache. This is usually done with the `IHI` instruction. If very large blocks of memory are to be written (for example, larger than the instruction cache size), it might be more efficient to invalidate the entire instruction cache (using the `III` instruction) rather than invalidating each cache line of memory that gets written. If the configuration supports writeback cache, the data cache must also be flushed to memory. This is usually done using the `DHWB` instruction.

Memory accesses can encounter bus errors (if the system supports it) or other exceptions (refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* or the *Xtensa Microprocessor Programmer's Guide*.). The host debugger should take particular attention to the `DSR.Exception` bit when executing load and store instructions, so that such conditions can be reported to the user if appropriate.

7.3.5.6 TIE State

TIE state is often accessed using the `RUR` and `WUR` instructions, in a manner very similar to special registers using the `RSR` and `WSR` instructions. However, if they access TIE state that is part of a co-processor, these instructions will generate a co-processor exception if the corresponding `CPENABLE` register bit is not set. In a low-level debugger (*e.g.*, used for kernel debugging) one typically needs to save and set the relevant `CPENABLE` bits before attempting to access the co-processor's state. In a task-level debugger (*e.g.*, when debugging at the application level using a kernel aware debugger), it is usually more appropriate to show the task-visible value for these registers, which might at any given time be saved in memory rather than be live in the co-processor. This could be done by letting the target OS handle any co-processor exception, or by accessing the co-processor state wherever it happens to be (in the co-processor or in memory). Note that depending on the OS, a task might not have any state for a given co-processor if it never uses that co-processor.

Some TIE states may be only accessible via special TIE instructions. For example, some TIE co-processors provide states that are only accessible using TIE load and store instructions. This requires the host debugger to have some region of memory allocated to it for accessing that state through memory. Issues relating to `CPENABLE` bits apply here as well if such TIE state is part of a TIE co-processor.

Some TIE states may even exist that are totally inaccessible (at least not directly) to the host debugger or to the target-side software. The possibilities are too involved to describe completely here. Refer to the particular TIE configured in a given Xtensa core.

Note that the Core HAL contains save and restore sequences for all TIE states designed to be context-switchable by an OS (see the *Xtensa System Software Reference Manual*).

Those looking to implement a generic (core configuration-independent) Xtensa debugger that fully supports access to TIE state should contact Cadence for more details on what is available to support them.

7.4 General Debugger Issues

This section covers general debugger issues unrelated to OCD.

7.4.1 Spilling Register Windows to the Stack

This section only applies when debugging code that uses windowed calling conventions. For more details on the stack layout for the windowed ABI, refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* or the *Xtensa Microprocessor Programmer's Guide*.

When a debug exception is taken, the register windows of callers of the current function may either be in the register file or have been spilled to the stack. The proper thing for the debugger to do when doing a call traceback or whenever looking at a calling function's address registers, is to examine the `WINDOWSTART` register bits (in combination with `WINDOWBASE`) to determine whether to look in the register file or on the stack for the relevant address register values.

Some debuggers, for simplicity, forcefully spill register windows to the stack whenever the code being debugged is stopped. This way they can always look at the stack for the contents of register windows other than the current one. Other than being more intrusive and accordingly less faithful to the actual execution of the code had it not stopped, this has a few implications. One is that stack spilling should only be done if `PS.WOE` is set and `PS.INTLEVEL` is zero. Another is that the delay in spilling the stack could be noticeable when done across a TAP. This is especially true of debuggers like GDB, that upon user request to single-step a C source code line, will sometimes single-step Xtensa instructions until the C source code line number changes. Depending on how the stack spilling logic is integrated with the debugger, the stack might end up being spilled (mostly redundantly) for each Xtensa instruction being single-stepped, greatly lowering performance.

One solution to this is to clear the WINDOWSTART bits of the register windows that have been spilled, so that they need not be spilled again. However there is one small twist to this solution, which is that the relevant WINDOWSTART bits must not be cleared if the current PC points to a RETW instruction. Part of RETW's functionality is to cause a window underflow if required, because the caller's window needs to be live for RETW to complete. If the relevant WINDOWSTART bits were cleared after spilling, the caller's window would no longer be live, so single-stepping RETW would never progress anywhere (unless perhaps code is being single-stepped using ICOUNTLEVEL=1).

7.4.2 Handling BREAK Instruction Conventions

An Xtensa debugger that relies on the debug exception mechanism to gain control of the processor, such as is typically done with OCD, can only gain control when a debug exception occurs (see Section 5.2). It does not automatically gain control upon level-1 exceptions for instance, which comprises most of an Xtensa processor's exception causes. Indeed, with OCD, one would not want the processor to stop on every level-1 exception. Level-1 interrupts, system calls, and other potentially high-frequency events generate level-1 exceptions, so making the processor stop and wait thousands or millions of cycles for the host debugger to make a determination through the TAP would make a system ridiculously slow.

Thus, only take control of selected level-1 exceptions (and even then, one might only want control of certain exceptions under specific conditions). The Xtensa debugger does not provide a hardware mechanism to direct specific level-1 exception causes to a debug exception, so this must be done in software. If every OS did this differently, a debugger would have to be very OS aware to achieve this, which is undesirable. So a set of conventions using the BREAK instruction was developed to allow an OS-independent debugger to intercept selected level-1 exceptions (and other exceptions) with the assistance of the OS. Basically, the OS executes a specific BREAK instruction, with specific guidelines and rules, for exceptions that the debugger should be allowed to intercept. This is described in greater detail in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

7.4.3 Executing OCD Instructions with Processor Stalls

Certain processor options, such as an inbound PIF request, can cause the processor to stall for large numbers of cycles. In these cases, the execution of processor instructions can take a large number of core clock cycles to complete. For example, you may need to poll the DSR many times before an operation or a debug interrupt initiated from the tap is complete.

A stall can also delay completion of the RFDO instruction. You must poll the DSR.ExecDone bit to confirm that the instruction has completed.

8. Performance Monitor for Xtensa NX Processors

Performance monitoring is a useful tool that allows software developers to profile code and SoC architects to make system design improvements. Key features include:

- Performance counters are accessible via the Access Port (Chapter 2) by software running on the Xtensa processor (using the ERI interface), and by external masters (using APB or JTAG interfaces)
- Easy to implement hardware profiling

8.1 Overview

There are a configurable number of counters with associated control and status registers. For example, the control registers select what is counted, whereas the status registers indicate whether an overflow occurred.

`PerfMonInt` is an interrupt that is asserted when any performance counter overflows. A single register records the PC when an overflow causes an interrupt.

The Performance Monitor registers are accessible as part of the Debug module register space. The configuration prerequisites for the Performance Monitor are the OCD option and Traceport.

If PerfMon is configured, certain signals are output as part of the Traceport, e.g. PDebugLSnStat. Refer to Table 16–78 in Section 16.2 “Basic Signals” for a complete list.

The number of performance counters selectable is 8 only.

A new “profiling” interrupt type appears on the Xtensa Xplorer Interrupts page. If selected, `PerfMonInt` is automatically connected to one of the Interrupt Controller inputs. This connection happens inside ScalarPipe and is not visible to the external system.

8.2 Architecture

The configurable number of 32-bit counters each have an identical range of possible inputs. Each performance counter has an associated control and status register that requires privileged access. There is also a PC register that records the PC (or the best known PC) at the time that the performance counter interrupt was asserted.

For compatibility with standard memory-mapped device conventions and usage, all registers are 32-bits wide.

Table 8–37. Performance Count Register Map

Addr	Register	Width	Access **	Reset Value	Description
0x1000	PMG	32	R/W	0x0	Global control/status for all performance counters
0x1010	INTPC	32	RO	0x0	PC at the cycle of the event that caused PerfMonInt assertion
0x1080 - 0x109C	PM0 to PM7	32	R/W	0x0	Performance counter value
0x1100 - 0x111C	PMCTRL0 to PMCTRL7	32	R/W	0x0	Performance counter control register
0x1180 - 0x119C	PMSTAT0 to PMSTAT7	32	R/clr	0x0	Performance counter status register

** See “Register Table Terms” on page xxii

8.2.1 Counter

A counter is a 32-bit wide register. Counters are not only readable, but are also made writable in the event that profiling software wants to be interrupted after a specific number of events have occurred. There are no restrictions for when a counter can be read or written. It is the responsibility of software not to corrupt a register value in the middle of counting.

8.2.2 Control Register

Each counter has an associated control register. The fields of the control register are tabulated in Table 8–38.

Table 8–38. Control Register Fields

Field	Bits	Access	Reset Value	Description
INTEN	0	R/W	0	Enables assertion of PerfMonInt output when overflow happens.
-	2:1	RO	0	Reserved. Read as 0.
KRNLCNT	3	R/W	0	Enables counting when EXECLEVEL > TRACESCOPE (i.e. If this bit is set, this counter counts only when EXECLEVEL >TRACESCOPE; if this bit is cleared, this counter counts only when EXECLEVEL ≤ TRACESCOPE).
TRACESCOPE	6:4	R/W	0	Compares this value to EXECLEVEL when deciding whether to count.

Table 8–38. Control Register Fields (continued)

Field	Bits	Access	Reset Value	Description
-	7	RO	0	Reserved. Read as 0.
SELECT	12:8	R/W	0	Selects input to be counted by the counter.
-	15:13	RO	0	Reserved, read as 0.
MASK	31:16	R/W	0	Selects input subsets to be counted (counter will increment only once even if more than one condition corresponding to a mask bit occurs).

EXECLEVEL is the Xtensa core's current execution level as reported by the Traceport. Refer to Section 8.8.2 for more details.

As Table 8–38 shows, not all MASK bits are used for each event type.

8.2.3 Status Register

Each counter has an associated status register. The fields of the status register are tabulated in Table 8–39.

Table 8–39. Status Register Fields

Field	Bits	Access	Reset Value	Description
OVFL	0	R/clr	0	Counter Overflow. Sticky bit set when a counter rolls over from 0xffffffff to 0x0
-	3:1	RO	0	Reserved
INTASRT	4	R/clr	0	This counter's overflow caused PerfMonInt to be asserted
-	31:5	RO	0	Reserved

Overflow is cleared by writing a 1'b1 to bit 0 of the Status register. Similarly, IntAsserted is cleared by writing a 1'b1 to bit 4 of the Status register. Note that clearing these bits does not disable counting. Section 8.3 describes a well-defined counting session.

8.2.4 Global Register

PMG is a single register for the entire Performance Monitor block. The fields of the PMG register are tabulated in Table 8–40.

Table 8–40. Global Register Fields

Field	Bits	Access	Reset Value	Description
PMEN	0	R/W	0	Overall enable for all performance counting
-	31:1	RO	0	Reserved

Setting PMEN enables counting, that is, all performance monitor functionality is gated by this enable. This bit is generally cleared when setting up counters. When all setup is completed, it can be set to start all counters simultaneously. Clearing this bit will stop all counting. As indicated in the table above, the reset value of PMEN is 0.

PMEN has another purpose, namely enabling clocks to the Debug module, and asserting the Traceport enable PDebugEnable.

There is a path in the hardware from PMEN to PDebugEnable to Xtensa core internals to which enabling the Traceport to event information accessible to the Performance Monitor. The path, which is several flops long, includes the processor pipeline. Therefore, PMEN must be enabled at least ten CLK cycles prior to an event of interest that you would like to count.

8.2.5 Interrupt Program Counter Register

INTPC is a single register for the entire Performance Monitor block. The fields of the INTPC register are tabulated in Table 8–41.

Table 8–41. PC Register Fields

Field	Bits	Access	Reset Value	Description
INTPC	31:0	RO	0	The PC (or the last-known good PC) at the time that the performance counter interrupt was asserted

8.3 A Performance Counting Session

The values of the PMEN bit and the PerfMonInt signal define various states of a normal counting session as shown in Table 8–42.

Table 8–42. Performance Counting Session States

State	PMEN	Perf MonInt	Description	Transition
SESS_SETUP	0	0	Count session setup: ■ all INTASRTs are clear by definition ■ INTPC has undefined value ■ all OVFLs should be cleared ■ PMCTRL fields should be set up ■ PM values must be written to if required	Setting PMEN causes transition to COUNTING
COUNTING	1	0	Normal counting: ■ all INTASRTs are clear ■ INTPC tracks current execution PC ■ OVFLs may happen if they are not programmed to cause INTASRT ■ PMCTRL fields would normally not be meddled with ■ PM values increment with events	An overflow causing INTASRT in turn causes transition to INT_TRIGD
INT_TRIGD	1	1	Counting continues with interrupt set: ■ one INTASRT is set ■ INTPC holds on to the PC associated with the first INTASRT ■ OVFLs may continue to happen ■ PMCTRL fields would normally not be meddled with ■ PM values increment with events	Clearing PMEN causes transition to DENOUEMENT
DENOUEMENT	0	1	End of counting: ■ one INTASRT is set ■ INTPC continues to hold on to PC associated with first INTASRT ■ no more OVFLs will happen ■ PMCTRL and PMSTAT fields, and the PM values would all be read, but normally not written	Clearing all INTASRTs causes transition to SESS_SETUP

Note that it is possible to go from the INT_TRIGD state to the COUNTING state by clearing all INTASRTs and OVFLs (and optionally updating PM values or even PMCTRL fields). However, this is not recommended because counting continues under these conditions, and therefore the window for the new counting is not cleanly defined. This is particularly true when using multiple counters for different events in the same time frame.

8.4 Hardware Details

The following sections provide details about the hardware.

8.4.1 Traceport Interface

Most of the performance information comes from the Traceport. TRAX is a co-user of Traceport information within the Debug module.

8.4.2 Performance Interrupt

`PerfMonInt` is an output pin of the Debug module that is asserted when any performance counter overflows. The PC register records the PC (or the best known PC) when an overflow causes an interrupt. While a given PC might not seem to have much value when doing profiling, the PC register does allow more accurate reporting of profiling information.

`PerfMonInt` is not brought out of Xtmem, it is a signal from the Debug module to the Interrupt Control module. It is recommended (but not required) to dedicate one priority level to reduce interrupt latency.

As with other interrupts, the profiling interrupt is maskable via the special register INTENABLE.

8.4.3 Multiple Overflows

If more than one overflow occurs prior to the processor's (or external agent's) ability to clear the interrupt, only the PC associated with the first counter to overflow is available. In other words, only the IntAsserted of the first counter is set. Writing that counter's IntAsserted bit with a 1'b1 clears the interrupt.

If multiple overflows happen in the same cycle, only one PC — the only available PC associated with the instruction or bubble — is recorded. All IntAsserted bits must be cleared in order to clear the interrupt. What happens in each scenario is detailed in Table 8–43.

Table 8–43. Multiple Overflows Prior to Ability to Clear Interrupt

Prior to any Overflow we get ➔	More than one Overflow in Successive Cycles	More than one Overflow in the Same Cycle
OVFL	of all the overflowing counters gets set – each in the cycle of its overflow	of all the overflowing counters gets set – in the cycle of the multiple overflows
INTASRT	of only the first counter gets set – in the cycle of the first overflow	of all the overflowing counters gets set – in the cycle of the multiple overflows
PerfMonInt signal	is asserted in the cycle of the first overflow	is asserted in the cycle of the multiple overflows
INTPC	contains the PC at the cycle of the first overflow-causing event	contains the PC at the cycle of the multiple overflow-causing events

8.4.4 Accessing the Counters

As described in Chapter 2, three independent mechanisms are provided to access the Debug module:

- JTAG
- APB
- ERI

Each interface has access to the same set of registers When viewed from any given interface, OCD, TRAX, and performance counter registers all occupy the same address space. When viewed from the APB or ERI, these three each live in separate 4 KB pages so that access privilege can be controlled on a page basis.

8.5 Count Overview

Table 8–44 defines the meaning of the Select and Mask bits in each Control Register. Each of the counters is capable of counting any of the events determined by this table. Events whose type is set to '1' in the mask are counted while events whose type is set to '0' in the mask are not.

Table 8–44. Select and Mask Meanings

Select Field	Description	Mask Bits (Sub-Events to Count)
0	Always Increment (Counts cycles)	Mask must be any non-zero value to enable count
1	Overflow of counter *n*-1 (Assume this is counter *n*.)	Mask must be any non-zero value to enable count
2	Successfully Retired Instructions	15: Non-branch instr (aka. non-CTI (control transfer instruction)) 14~12: Reserved 11: Last inst of loop and execution falls through to LEND (aka. loopback fallthrough) 10: Last inst of loop and execution transfers to LBEG (aka. loopback taken) 9: Reserved 8: Loop instr where execution falls into loop (aka. taken loop) 7: Conditional branch instr where execution falls through (aka. not-taken branch) 6: CALLn instr 5: J instr 4: Conditional branch instr where execution transfers to the target (aka. taken branch), or loopgtz/loopnez instr where execution skips the loop (aka. not-taken loop) 3: supervisor return instr i.e. RFDE, RFE, RFI, RFWO, RFWU 2: call return instr i.e. RET, RETW 1: CALLXn instr 0: JX instr
3	Data-related Stall cycles	15~5: reserved 4: Replay_C, StallR1toM and other LS stalls (including L1 DCache-Misses delineated below) 3: L1 Vector DCache-Miss from LS1 (if configured) 2: L1 Vector DCache-Miss from LS0 1: L1 Scalar DCache-Miss from LS1 (if configured) 0: L1 Scalar DCache-Miss from LS0
4	Instruction-related and Other Stall cycles	15~3: reserved 2: Uncached Instruction Fetch 1: L1 ICache Miss 0: Other

Table 8–44. Select and Mask Meanings (continued)

Select Field	Description	Mask Bits (Sub-Events to Count)
5	Exceptions and Pipeline Replays	15~8: reserved 7: Other exceptions 6: reserved 5: Window exception 4:reserved 3: reserved 2: Interrupt 1: reserved 0: Other Pipeline Replay (i.e. excludes cache miss etc.)
6	Hold and Other Bubble cycles	15~10: reserved 9: Bubbles other than DCache-misses, exceptions and those below 8: reserved 7: CTI (control transfer instruction) bubble (e.g. branch delay slot) 6: Pipeline bubble caused by a resource dependency such as register, cache (IHI/III/LICT/LICW/SICT/SICW instructions) and multicycle (resulting in R3 replays) 5-0: reserved
7	Instruction TLB Accesses (per instruction retiring)	15~4: reserved 3: ITLB Miss Exception 2: Hardware-assisted TLB Refill completes 1: Replay of instruction due to ITLB miss 0: ITLB Hit
8	Instruction Memory Accesses (per instruction retiring)	15~5: reserved 4: other sources than those below 3: reserved 2: All InstRAM accesses 1: reserved 0: Instruction Cache Hit
9	Data TLB Accesses	15~4: reserved 3: DTLB Miss Exception 2: Hardware-assisted TLB Refill completes 1: Replay of load/store due to DTLB miss 0: DTLB Hit

Table 8–44. Select and Mask Meanings (continued)

Select Field	Description	Mask Bits (Sub-Events to Count)
10 (LS0)[1] 16 (LS1)	L1S Data Memory Load Instruction	15~4: reserved 3: uncached load 2: Load from local memory i.e. DataRAM, DataROM, InstRAM 1: Data Cache Miss 0: Data Cache Hit
11 (LS0)[1] 17 (LS1)	L1S Data Memory Store Instruction	15~4: reserved 3: AXI Store 2: Store to local memory i.e. DataRAM, InstRAM 1: Data Cache Miss 0: Data Cache Hit
12 (LS0)[1] 18 (LS1)	L1V Data Memory Load Instruction	15~4: reserved 3: reserved 2: reserved 1: Data Cache Miss 0: Data Cache Hit
13 (LS0)[1] 19 (LS1)	L1V Data Memory Store Instruction	15~4: reserved 3: reserved 2: reserved 1: Data Cache Miss 0: Data Cache Hit
14 (LS0)[1] 20(LS1)	Scalar Data Memory Accesses (Load, Store, etc instructions)	15~4: reserved 3: Hit Modified 2: Hit Exclusive 1: Hit Shared 0: Cache Miss
15 (LS0)[1] 21(LS1)	Vector Data Memory Accesses (Load, Store, etc instructions)	15~4: reserved 3: Hit Modified 2: Hit Exclusive 1: Hit Shared 0: Cache Miss
27	iDMA	15~2: reserved 1: active cycles (channel 1) 0: active cycles (channel 0)

Table 8–44. Select and Mask Meanings (continued)

Select Field	Description	Mask Bits (Sub-Events to Count)
28	Instruction Length	15: reserved 14: 128-bit 13: 120-bit 12: 112-bit 11: 104-bit 10: 96-bit 9: 88-bit 8: 80-bit 7: 72-bit 6: 64-bit 5: 56-bit 4: 48-bit 3: 40-bit 2: 32-bit 1: 24-bit 0: 16-bit

Table 8–44. Select and Mask Meanings (continued)

Select Field	Description	Mask Bits (Sub-Events to Count)
29	Prefetch	15~8: reserved 7: D Prefetch AXI reads 6: LS1 DCache miss and D prefetch-buffer-lookup miss 5: LS0 DCache miss and D prefetch-buffer-lookup miss 4: LS1 DCache miss and D prefetch-buffer-lookup hit 3: LS0 DCache miss and D prefetch-buffer-lookup hit 2: I Prefetch AXI reads 1: ICache miss and I prefetch-buffer-lookup miss 0: ICache miss and I prefetch-buffer-lookup hit
30	Multiple Load/Store	15:10 reserved 9: 3 stores and 0 loads 8: 0 stores and 3 loads 7: 2 stores and 1 loads 6: 1 stores and 2 loads 5: 2 stores and 0 loads 4: 0 stores and 2 loads 3: 1 stores and 1 loads 2: 1 stores and 0 loads 1: 0 stores and 1 loads 0: 0 stores and 0 loads
31	Castout	15~4: reserved 3: reserved 2: Initiated by prefetch cache fill 1: Initiated by Load/Store unit 1 cache miss 0: Initiated by Load/Store unit 0 cache miss

1. Unless a processor is configured with multiple load/store units, only the select value marked LS0 is used. Specifically, select values 10 through 15 are used for load/store unit 0; select values 16 through 21 for load/store unit 1.

A counter will only increment by one even if more than one condition corresponding to a set mask bit occurs.

8.5.1 Configurability

The counted events shown in Table 8–44 are subject to configuration restrictions. For example, in configurations with:

- No ICache, configuring a counter to count only ICache miss (select = 8, mask bit 1)
- A single load/store unit, configuring a counter to count LS1 cache miss or hit from the scalar pipe (select = 16, mask bits 0 and 1 = 0x0003)

In these cases, the count value will remain unchanged, and `PerfMonInt` is never asserted.

8.5.2 Illogical Use of Counters

In general, there are no hardware additions to catch or ensure that software does not do something misleading. For example, consider a specific case where a Performance Monitor is configured to count L1S LS0 DCache miss stall cycles (select = 3, mask bit 0); that would typically be for say 15 cycles/miss, at the 7th cycle of the miss stall, from the APB interface, the control register is changed to count CTI (control transfer instruction) bubbles (select = 6, mask bit 7). The performance counter will behave as programmed, in this case it will count up to 7 and then simply stop, or if there are subsequent CTI bubbles, count these from 8 onwards.

8.5.3 Accuracy of Counts

The accuracy of Performance Monitor counts depends on the accuracy of the indicated Traceport events.

For events related to the completion of a valid instruction, such as JX instruction count (select = 2, mask bit 0), there is a guaranteed cycle available for the Xtensa core to output the appropriate indication on the Traceport. The information related to the instruction, such as PC, is also unambiguously available. For this reason, we guarantee 100% accuracy in counting such events.

For events not related to the completion of a valid instruction, such as L1S LS0 Data Cache Misses (select = 10, mask bit 1), there may not be a guaranteed cycle available to output the appropriate indication on the Traceport. This is due to the speculative nature of the processor pipeline and the fact that multiple events could be happening concurrently. Moreover, the information associated with the event, such as PC, may not be unambiguous. While it is high, we say that accuracy in counting such events is not 100%.

8.6 Event Details

This section provides more details about what is being counted for the different Select and Mask values of the control registers. The information is not always directly available; in many cases, the Performance Monitor needs logic to decode the event from what is being presented on the Traceport. This is referred to as the event-generating logic.

A note on terminology: “Retiring” (also known as “committing”) refers to an instruction reaching the W stage without being killed. Once an instruction reaches the W stage, its effects on architectural state cannot be reversed. Depending on the instruction, it may continue to update both architectural (e.g., long-latency user-TIE instructions) and microarchitectural states (e.g. DCache refill) beyond the W stage. In general, the performance monitor is unable to count events associated with post-W-stage effects.

8.6.1 Overflow of Lower Counter

As Select value 1 in Table 8–44 shows, one of the events selectable by counter *n* is the overflow of counter *n*-1. This means that when the user sets up counter *n-1* with a given select value and specifies select 1 for the next counter i.e. counter *n*, counter *n* is used to count the overflows of counter *n*-1. This allows concatenation of the two counters to count the performance events selected in the first counter. In this way it is possible to count more than 2^{32} occurrences of a given event.

Counts wider than even 2^{64} are possible by setting more than one adjacent counter with select = 1.

When counter 0 has select = 1, it counts the overflows of the largest numbered counter of the configuration. In other words, modulo arithmetic is used in determining *n*-1.

8.6.2 Retired Instructions

With select = 2, it is possible to get a dynamic count of successfully completing instructions. If all the masks are clear, the counter is off. If all masks are set, the counter counts ALL instructions retired.

If the masks are appropriately set, it is possible to count many types of instructions, such as the static number of unconditional loops (mask bit 8), dynamic number of loop iterations (mask bit 10), fall-through branches (mask bit 7), taken branches (mask bit 4), indexed jump/call (mask bits 5 and 6), returns (mask bit 2), etc.

Notice that the last instruction of a loop (aka. loopback taken and loopback fallthrough) masks are special in that they can happen on an ordinary (i.e., non control-transfer instruction). In these cases, there is no non-CTI (control transfer instruction) count; that is, they would not be counted if only mask bit 15 were set.

8.6.3 Data-related Stall Cycles

When select = 3 it is possible to count the processor data-related R stage stall cycles including those associated with L1 DCache misses from the Vector and Scalar pipelines. Stall cycles from all data-related sources can be selected for counting by mask bit 4. Specific DCache miss stall cycles from a particular pipeline (Vector or Scalar) and LS unit (0 or 1) can be selected for counting by mask bits 3:0. The stall cycles are the bubble cycles after the first one. The first bubble cycle is reported with select = 6 as defined in Section 8.6.6

8.6.4 Instruction-related and other Stall Cycles

When select = 4 it is possible to count the processor data-related R stage stall cycles associated with ICache misses and other non Data-related stalls. Particular stall causes can be selected for counting by mask bits 2:0. The stall cycles are the bubble cycles after the first one. The first bubble cycle is reported with select = 6 as defined in Section 8.6.6.

8.6.5 Exceptions and Replays

With select = 5, it is possible to count interrupts, exceptions, and replays, as follows:

- Mask bit 8 counts hardware-corrected memory errors in configurations that have ECC. Note that these do not result in as exception, i.e., no jump to a vector. The errors are corrected on the fly.
- Mask bit 7 counts exceptions other than those covered in other mask bits of this Select field (window exceptions and interrupts). Examples of this include memory errors (which again is separate from the hardware-corrected memory errors of mask bit 8), bus errors, illegal instructions, syscall, instruction-fetch-related errors, load/store-related errors, TLB-related errors, co-processor disabled, double exceptions, etc.
- Mask bit 5 corresponds to a window overflow or underflow exception.
- Mask bit 2 corresponds to interrupts.

- Mask bit 0 corresponds to replays. A replay could happen for a number of reasons. In many cases, replays happen in the same cycle that other events are signaled on the Traceport. Specifically, these are:
  - DCache miss
  - TLB miss
  - exception or interrupt
  - Hardware-assisted TLB refill
  - Hardware-corrected memory error

In other words, when counting replays using select = 5 and mask bit 0, it is important to remember that only replays that happen at cycles not coinciding with the above events are counted. The events in the list above can be counted using other select/mask values, and always result in replays.

8.6.6 Hold and Other Bubble Cycles

With select = 6, it is possible to count hold and other bubble cycles. A bubble happens when there is no valid instruction retiring in a cycle. A hold is one of the possible reasons for a bubble and occurs due to a pipeline conflict. Other causes of bubbles include branching or pipeline flushing.

Hold and other bubble counting is based on the Traceport reporting of these events, that is, register dependency, resource conflict, and control transfer. Bubbles are reported as encodings of `PDebugStatus` when `PDebugInst` indicates that there is no valid instruction retiring. Being a subcategory of bubbles, holds are reported on `PDebugData` as a bit map, and when `PDebugStatus` has a specific encoding (0000_10) that says that the bubble is because of a hold. Bubble and hold cycles that are countable are:

- Mask bit 9 counts bubbles for causes other than DCache-misses, exceptions, CTI (control transfer instruction), hold, and PSO derived from `PDebugStatus=1111_00`.
- Mask bit 7 counts bubbles due to a CTI instruction (e.g., branch delay slot) derived from `PDebugStatus=0001_00`.

- Mask bit 6 counts bubbles due to R-stage hold dependencies derived from `PDebugStatus=0000_10`.
  - MEMW, EXTW, or EXCW instruction dependency. Upon executing a barrier instruction (i.e. MEMW, EXTW, or EXCW) the pipeline is held to wait for prior instructions (e.g. stores) to complete.
  - Register dependencies or resource conflicts. These holds are bubbles due to register dependencies between instructions or conflicts for resources such as TIE ports.
  - Holds or bubbles that wait for various LS unit dependencies, e.g. cache access instructions (IHI/III/LICT/LICW/SICT/SICW), etc. to be resolved.
  - Other hold dependencies resulting from data or resource dependencies such as multicycle, which results in R3-stage replays.

8.6.7 Instruction Memory Accesses

Following are the instruction fetch events:

- Cache Miss: An instruction cache miss occurred, and a request was sent to the AXI.
- Cache Hit: An instruction in the W-stage is valid and is executing out of the cache.
- InstRAM: An instruction in the W-stage is valid and is executing out of one of the InstRAMs or L0 buffers.
- Uncached: Fetch of an instruction from bypass space.

Instruction cache miss is processed early in the pipeline, and is counted when the missing instruction would have been in the W-stage. An ICache miss will be counted once as a miss and once as a hit when the fill completes and the instruction successfully executes. When fetch is happening from bypass-attribute memory and the instruction is unaligned, two bypass fetches may be required before the instruction can execute.

8.6.8 Loads

Following are the load events:

- Cache Load: There is a retiring load instruction in the W-stage that is using the cache.
- Load Miss: There is a load in the W-stage that is meant to use the cache but is going to get data over the AXI.
- DataRAM Load: There is a retiring load instruction in the W-stage that is from local RAM (i.e. DataRAM).
- Load from regions with attribute “Bypass” or “No Allocate”.

L32EX instructions are counted as part of the load categories.

8.6.9 Stores

Following are the store events:

- Cached Store: There is a retiring store instruction in the W-stage that is using the cache.
- Store Miss: There is a load in the W-stage that is meant to use the cache but is going to get data over the AXI. In write-back configurations, this may also entail a cache refill.
- DataRAM Store: There is a retiring store instruction in the W-stage that is to local RAM – i.e. DataRAM.
- AXI Store: There is a retiring store in the W-stage that will be sent to the AXI (write-through, bypass, or no-allocate attribute). It may also be sent to the cache if the data is in the cache (write-through hit, no allocate hit).

S32EX instructions are counted as part of the store categories. They are counted simply when/if they retire, regardless of the result of the compare, that is, without regard to whether the write to the target memory happened.

8.6.10 Data Memory Accesses

These are events related to any load/store/S32EX/L32EX instructions.

- Cache Miss: Load/store in W-stage that was supposed to hit in the cache but did not.
- Hit Shared: Load/store in W-stage that hits a line that is shared. A load retires, but store is killed and prompts a refill.
- Hit Exclusive: Load/store in W-stage that hits a line that is exclusive.
- Hit Modified: Load/store in W-stage that hits a line that is modified.

8.6.11 IDMA Activity

With select = 27 iDMA activity cycles can be counted from the iDMA control state inputs (IDMAIdle[1:0] for two iDMA channels) it receives for each iDMA channel (up to 2). The iDMA Busy state encoding is for when DMA operations are ongoing. Counting cycles in the iDMA Busy state gives the number of activity cycles for each iDMA channel.

8.6.12 Instruction Length

With select = 28, it's possible to count the number of instructions executed that are of a particular length. PDebugInst[7:0] encodes the number of bytes of the instruction length. Subtracting 2 from this value will generate the Mask encodings for Select 28 in Table 8–44.

8.6.13 Prefetch

Prefetch effectiveness can be deduced by comparing cache hit and miss counts (select = 8 for instructions and select = 15 and 21 for L1V data) for different prefetch strategies. In addition, with select = 29 specific prefetch statistics can be obtained for use to determine prefetch effects.

Cache misses with prefetch buffer lookup misses (select = 29, masks 1,5 and 6, to count IPrefLookupMIss, DPrefLookupMiss0, and DPrefLookupMiss1, respectively) show when prefetch is not helping.

Cache misses with Prefetch buffer lookup hits (select = 29, masks 0,3 and 4, to count IPrefLookupHit, DPrefLookupHit0, and DPrefLookupHIt1, respectively) show when prefetch is helping, but not early enough to get the data to cache in time for when it is requested.

The number of Prefetch AXI reads can be counted with select = 29, masks 2 and 7 (IPrefFillToL1 and DPrefFillToL1 signals).

8.6.14 Multiple Load/Store

With select = 30, it is possible to count multiple load/stores.

8.6.15 Castout

Select = 31 can be used to count the number of DCache castouts initiated from each of the LS units (mask bits 0 and 1, to count CastoutLS0 and CastoutLS1, respectively) and from prefetch DCache fills (mask bit 2 to count CastoutPrefetch).

8.7 Profiling Code with the Performance Monitor

The Performance Monitor allows the user to profile code. However, to get certain code-profiling metrics, especially those that are similar to the metrics provided by the ISS, it might be necessary to collect and combine multiple performance counts. An example where different performance counts need to be combined to get the full overhead cycle count is data cache miss penalty in LS0.

The first is a replay penalty. This starts at the W stage of the load/store that would have retired, but is not because it gets the DCache miss. In this cycle there is a replay, and it is indicated by `PDebugLSnState` of the Traceport. [This is also indicated by `PDebugStatus`[5:0]=6'b001_00 or `PDebugStatus`[5:0]=6'b0011_10 depending on if it is an L1S or L1V cache miss.] For example, to count L1S DCache load misses, set select=10, mask bit 1.

In order to process the miss, R stage stall is asserted (Traceport has PDebugInst[7:0] =8'b0 and PDebugStatus[5:0]=6'b1000_00), and the relevant causes are countable with select=3 and mask bit 0 and/or 2 for L1S and/or L1V DCache miss.

Also, as part of processing the miss it might be necessary for the processor to do an R-stage hold of the replaying load. This is indicated on the Traceport by the combination of PDebugInst[7:0]=8'b0, PDebugStatus[5:0]=6'b0000_10. To count this, set select=6 and mask bit 6.

In summary, each of the components of a DCache miss are individually counted using separate counters. Since it is necessary to use a recipe that combines various counts appropriately, the number of counters selected at processor-configure time must be considered carefully.

Following are recipes for some metrics of common interest.

8.7.1 Retired Instructions

To count all committed instructions reaching the W-stage, use select = 2 and set all mask bits i.e. a mask value of 0xffff.

8.7.2 Branch Penalty

Branch penalties may be counted using select = 6 and mask bit 7 i.e. a mask value of 0x0080.

8.7.3 Pipeline and Resource Dependencies

R-stage holds due to pipeline and resource dependencies, may be counted using select = 6 and mask bit 6 i.e. a mask value of 0x0040.

8.7.4 Instruction Cache Miss Penalty

To count the instruction cache miss penalty, meaning bubble cycles waiting for an instruction fetch, use select = 4 and mask bit 1 i.e. value of 0x0002.

8.7.5 Data Cache Miss Penalty

To count the data cache miss penalty, meaning bubble cycles waiting for load or store miss to be processed, use select = 3 and mask value of 0x000f. If LS1 is configured the data cache miss penalty can be separately delineated for each LS unit with a mask value of 0x000a for LS1 and 0x0005 for LS0.

8.8 Software

Following are software tools- and OS-related notes regarding the use of performance counters.

8.8.1 Cadence-Provided Profiler

Cadence provides software tools and methods to analyze a program run using performance counters. These are described in the Hardware-Based Performance Analysis chapter of the *Xtensa Software Development Toolkit User's Guide*.

8.8.2 Operating System Relationship with EXECLEVEL

The Xtensa processor's current execution level (EXECLEVEL) is a value reported on the Traceport that classifies the code currently running in categories useful to track for profiling or tracing purposes, such as: application thread, exception handler, interrupt handler, or dispatch code. For example, this allows performance counters to selectively count at specified execution levels, using TRACESCOPE and KRNLCNT fields described in Section 8.2.2. Refer to Table 16–86 in the Traceport chapter of this guide for the encoding of EXECLEVEL on Traceport signals PDebugInst[19:17].

Note: When using the Windowed ABI, register window overflow and underflow exception handling (part of interruptible dispatch) does not change the execution level.

In order for the reported EXECLEVEL to be meaningful, the OS must follow a simple convention for setting PS.Stk at startup, according to how it manages stacks. The following describes how this is generally done for single-thread and multi-threaded runtimes.

Single-threaded runtimes (such as XTOS) that operate on a single interrupt stack must initialize PS.Stk to FirstInt in startup code, before calling any function. This allows distinguishing the application from any other handler, which will run with PS.Stk = Interrupt. EXECLEVEL values are as shown in Table 8–45.

Table 8–45. Traceport EXECLEVEL for Single-Threaded Application

EXEC LEVEL	Execution State	On Which Stack	PS.Stk	Interrupt Active
101	Non-interruptible dispatch code	n/a	–	–
100	Interrupt handler (or exception in interrupt handler)	interrupt	Interrupt	yes
011	Exception handler (for exception in application)	interrupt	Interrupt	no
010	Application	interrupt	FirstInt	– (no)

Multi-threaded runtimes (such as most RTOS) that operate with a separate kernel stack for each thread but a common interrupt stack for interrupts, must initialize PS.Stk to FirstKer for each thread's startup. This allows distinguishing thread code from other handlers—thread exception handlers will run with PS.Stk = Kernel, and interrupts with PS.Stk = Cross or Interrupt. EXECLEVEL values are as shown in Table 8–46.

Table 8–46. Traceport EXECLEVEL for Multi-Threaded Application

EXEC LEVEL	Execution State	On Which Stack	PS.Stk	Interrupt Active
101	Non-interruptible dispatch code	n/a	–	–
100	Interrupt handler (or exception in interrupt handler)	interrupt	Cross or Interrupt	yes
011	Exception handler (for exception in thread)	kernel	Kernel	– (no)
010	Application thread	kernel	FirstKer	– (no)

9. Performance Monitor for Xtensa LX Processors

Performance monitoring is a useful tool that allows software developers to profile code and SoC architects to make system design improvements. Key features include:

- Performance counters are accessible via the Access Port (Chapter 2) by software running on the Xtensa processor (using the ERI interface), and by external masters (using APB or JTAG interfaces)
- Easy to implement hardware profiling
- Resides in the Debug module, which belongs to a separate PSO domain if PSO is appropriately configured, so chips in the field may power down the logic

9.1 Overview

There are a configurable number of counters with associated control and status registers. For example, the control registers select what is counted, whereas the status registers indicate whether an overflow occurred.

`PerfMonInt` is an interrupt that is asserted when any performance counter overflows. A single register records the PC when an overflow causes an interrupt.

The Performance Monitor registers are accessible as part of the Debug module register space. The configuration prerequisites for the Performance Monitor are the OCD option and Traceport.

If the Performance Monitor is configured, certain signals are output as part of the Traceport, e.g. PDebugLSnStat. Additional signals are output if other functionality is configured, e.g. PDebugInbPIF if Inbound PIF is selected. Refer to Table 15–62 in Section 15.2 “Basic Signals” for a complete list.

The number of performance counters selectable is 2, 4, or 8; for the remainder of this chapter this is referred to generically as *n*.

A new “profiling” interrupt type appears on the Xtensa Xplorer Interrupts page. If selected, `PerfMonInt` is automatically connected to one of the `BInterrupt` bits. This connection happens inside Xttop and is not visible to the external system.

9.2 Architecture

The configurable number of 32-bit counters each have an identical range of possible inputs. Each performance counter has an associated control and status register that requires privileged access. There is also a PC register that records the PC (or the best known PC) at the time that the performance counter interrupt was asserted.

For compatibility with standard memory-mapped device conventions and usage, all registers are 32-bits wide.

Table 9–47. Performance Count Register Map

Addr	Register	Width	Access **	Reset Value	Description
0x1000	PMG	32	R/W	0x0	Global control/status for all performance counters
0x1010	INTPC	32	RO	0x0	PC at the cycle of the event that caused PerfMonInt assertion
0x1080 - 0x109C	PM0 to PM*n**	32	R/W	0x0	Performance counter value.
0x1100 - 0x111C	PMCTRL0 to PMCTRL*n**	32	R/W	0x0	Performance counter control register
0x1180 - 0x119C	PMSTAT0 to PMSTAT*n**	32	R/clr	0x0	Performance counter status register

* Note: *n* refers to the number of performance counters minus one ** See “Register Table Terms” on page vi

9.2.1 Counter

A counter is a 32-bit wide register. Counters are not only readable, but are also made writable in the event that profiling software wants to be interrupted after a specific number of events have occurred. There are no restrictions for when a counter can be read or written. It is the responsibility of software not to corrupt a register value in the middle of counting.

9.2.2 Control Register

Each counter has an associated control register. The fields of the control register are tabulated in Table 9–48.

Table 9–48. Control Register Fields

Field	Bits	Access	Reset Value	Description
INTEN	0	R/W	0	Enables assertion of PerfMonInt output when overflow happens
KRNLCNT	3	R/W	0	Enables counting when CINTLEVEL* > TRACELEVEL (i.e. If this bit is set, this counter counts only when CINTLEVEL >TRACELEVEL; if this bit is cleared, this counter counts only when CINTLEVEL ≤ TRACELEVEL)
TRACELEVEL	7:4	R/W	0	Compares this value to CINTLEVEL* when deciding whether to count
SELECT	12:8	R/W	0	Selects input to be counted by the counter
-	15:13	RO	0	Reserved, read as 0
MASK	31:16	R/W	0	Selects input subsets to be counted (counter will increment only once even if more than one condition corresponding to a mask bit occurs)

* Note: CINTLEVEL is the Xtensa core's current interrupt level as defined in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*, and here specifically, as reported by Traceport signals PDebugInst[27:24].

As Table 9–49 shows, not all MASK bits are used for each event type.

9.2.3 Status Register

Each counter has an associated status register. The fields of the status register are tabulated in Table 9–49.

Table 9–49. Status Register Fields

Field	Bits	Access	Reset Value	Description
OVFL	0	R/clr	0	Counter Overflow. Sticky bit set when a counter rolls over from 0xffffffff to 0x0.
-	3:1	RO	0	Reserved
INTASRT	4	R/clr	0	This counter's overflow caused PerfMonInt to be asserted.
-	31:5	RO	0	Reserved

Overflow is cleared by writing a 1'b1 to bit 0 of the Status register. Similarly, IntAsserted is cleared by writing a 1'b1 to bit 4 of the Status register. Note that clearing these bits does not disable counting. Section 9.3 describes a well-defined counting session.

9.2.4 Global Register

PMG is a single register for the entire Performance Monitor block. The fields of the PMG register are tabulated in Table 9–50.

Table 9–50. Global Register Fields

Field	Bits	Access	Reset Value	Description
PMEN	0	R/W	0	Overall enable for all performance counting
-	31:1	RO	0	Reserved

Setting PMEN enables counting, that is, all performance monitor functionality is gated by this enable. This bit is generally cleared when setting up counters. When all setup is completed, it can be set to start all counters simultaneously. Clearing this bit will stop all counting. As indicated in the table above, the reset value of PMEN is 0.

PMEN has another purpose, namely enabling clocks to the Debug module, and asserting the Traceport enable PDebugEnable.

There is a path in the hardware from PMEN to PDebugEnable to Xtensa core internals, which enables the Traceport to event information accessible to the Performance Monitor. The path, which is several flops long, includes the processor pipeline. Therefore, PMEN must be enabled at least ten CLK cycles prior to an event of interest that you would like to count.

9.2.5 Interrupt Program Counter Register

INTPC is a single register for the entire Performance Monitor block. The fields of the INTPC register are tabulated in Table 9–51.

Table 9–51. PC Register Fields

Field	Bits	Access	Reset Value	Description
INTPC	31:0	RO	0	The PC (or the last-known good PC) at the time that the performance counter interrupt was asserted

9.3 A Performance Counting Session

The values of the PMEN bit and the PerfMonInt signal define various states of a normal counting session as shown in Table 9–52.

Table 9–52. Performance Counting Session States

State	PMEN	Perf MonInt	Description	Transition
SESS_SETUP	0	0	Count session setup: ■ all INTASRTs are clear by definition ■ INTPC has undefined value ■ all OVFLs should be cleared ■ PMCTRL fields should be set up ■ PM values must be written to if required	Setting PMEN causes transition to COUNTING
COUNTING	1	0	Normal counting: ■ all INTASRTs are clear ■ INTPC tracks current execution PC ■ OVFLs may happen if they are not programmed to cause INTASRT ■ PMCTRL fields would normally not be meddled with ■ PM values increment with events	An overflow causing INTASRT in turn causes transition to INT_TRIGD
INT_TRIGD	1	1	Counting continues with interrupt set: ■ one INTASRT is set ■ INTPC holds on to the PC associated with the first INTASRT ■ OVFLs may continue to happen ■ PMCTRL fields would normally not be meddled with ■ PM values increment with events	Clearing PMEN causes transition to DENOUEMENT
DENOUEMENT	0	1	End of counting: ■ one INTASRT is set ■ INTPC continues to hold on to PC associated with first INTASRT ■ no more OVFLs will happen ■ PMCTRL and PMSTAT fields, and the PM values would all be read, but normally not written	Clearing all INTASRTs causes transition to SESS_SETUP

Note that it is possible to go from the INT_TRIGD state to the COUNTING state by clearing all INTASRTs and OVFLs (and optionally updating PM values or even PMCTRL fields). However, this is not recommended because counting continues under these conditions, and therefore the window for the new counting is not cleanly defined. This is particularly true when using multiple counters for different events in the same time frame.

9.4 Hardware Details

The following sections provide details about the hardware.

9.4.1 Traceport Interface

All of the performance information comes from the Traceport, which is an input to the Debug module. TRAX is a co-user of Traceport information within the Debug module. Traceport changes from previous releases that are pertinent to the Performance Monitor includes:

- `PDebugLS`*n*`Stat` increases in width to include information such as decoded operand size and TLB hit/miss
- More detailed information on the type of exception, for example, LoadStoreTLBMissCause
- Instruction TLB hit information

9.4.2 Performance Interrupt

`PerfMonInt` is an output pin of the Debug module that is asserted when any performance counter overflows. The PC register records the PC (or the best known PC) when an overflow causes an interrupt. While a given PC might not seem to have much value when doing profiling, the PC register does allow more accurate reporting of profiling information.

`PerfMonInt` is a signal from the Debug module to the Xtensa core alone, and not brought out of Xtmem. To facilitate use of Cadence-supplied profiling software, Xtensa Xplorer automatically connects `PerfMonInt` to one of the `BInterrupt` pins based on the new “profiling” interrupt type. It is recommended (but not required) to dedicate one level to reduce interrupt latency.

As with other interrupts, the profiling interrupt is maskable via the special register INTENABLE.

9.4.3 Multiple Overflows

If more than one overflow occurs prior to the processor’s (or external agent’s) ability to clear the interrupt, only the PC associated with the first counter to overflow is available. In other words, only the IntAsserted of the first counter is set. Writing that counter’s IntAsserted bit with a 1’b1 clears the interrupt.

If multiple overflows happen in the same cycle, only one PC—the only available PC associated with the instruction or bubble—is recorded. All IntAsserted bits must be cleared in order to clear the interrupt.

Table 9–53. Multiple Overflows Prior to Ability to Clear Interrupt

Prior to any Overflow we get ➔	More than one Overflow in Successive Cycles	More than one Overflow in the Same Cycle
OVFL	of all the overflowing counters gets set – each in the cycle of its overflow	of all the overflowing counters gets set – in the cycle of the multiple overflows
INTASRT	of only the first counter gets set – in the cycle of the first overflow	of all the overflowing counters gets set – in the cycle of the multiple overflows
PerfMonInt signal	is asserted in the cycle of the first overflow	is asserted in the cycle of the multiple overflows
INTPC	contains the PC at the cycle of the first overflow-causing event	contains the PC at the cycle of the multiple overflow-causing events

9.4.4 Accessing the Counters

As described in Chapter 2, three independent mechanisms are provided to access the Debug module:

- JTAG
- APB
- ERI

Each interface has access to the same set of registers. When viewed from any given interface, OCD, TRAX, and performance counter registers all occupy the same address space. When viewed from the APB, these three each live in separate 4 KB pages so that access privilege can be controlled on a page basis. The view from JTAG or ERI is analogous for consistency.

9.5 Count Overview

Table 9–54 defines the meaning of the Select and Mask bits in each Control Register. Each of the counters is capable of counting any of the events determined by this table. Events whose type is set to '1' in the mask are counted while events whose type is set to '0' in the mask are not.

Table 9–54. Select and Mask Meanings

Select Field	Description	Mask Bits (Sub-Events to Count)
0	Always Increment (Counts cycles)	Mask must be any non-zero value to enable count
1	Overflow of counter *n*-1 (Assume this is counter *n*.)	Mask must be any non-zero value to enable count
2	Successfully Retired Instructions	15: Non-branch instr (aka. non-CTI) 14~12: Reserved 11: Last inst of loop and execution falls through to LEND (aka. loopback fallthrough) 10: Last inst of loop and execution transfers to LBEG (aka. loopback taken) 9: Reserved 8: Loop instr where execution falls into loop (aka. taken loop) 7: Conditional branch instr where execution falls through (aka. not-taken branch) 6: CALLn instr 5: J instr 4: Conditional branch instr where execution transfers to the target (aka. taken branch), or loopgtz/loopnez instr where execution skips the loop (aka. not-taken loop) 3: supervisor return instr i.e. RFDE, RFE, RFI, RFWO, RFWU 2: call return instr i.e. RET, RETW 1: CALLXn instr 0: JX instr
3	Data-related GlobalStall cycles	15~10: reserved 8: Bank-conflict stall 7: Uncached load stall (included in miss-handling stall below) 6: Miss-handling stall 5: Data inbound-PIF request stall (includes s32c1i) 4: Data RAM/ROM/XLMI busy stall 3: DCache-miss stall 2: Store buffer conflict stall 1: Store buffer full stall 0: Reserved

Table 9–54. Select and Mask Meanings (continued)

Select Field	Description	Mask Bits (Sub-Events to Count)
4	Instruction-related and Other GlobalStall cycles	15~9: reserved 8: Iterative divide stall 7: Iterative multiply stall 6: FastL32R stall 5: Uncached fetch stall 4: External RunStall signal status 3: TIE port stall 2: Instruction RAM inbound-PIF request stall 1: Instruction RAM/ROM busy stall 0: ICache-miss stall
5	Exceptions and Pipeline Replays	15~9: reserved 8: HW-corrected memory error 7: Other exceptions 6: Allocate exception 5: Window exception 4: NMI 3: Debug exception 2: Greater-than-level-1 interrupt 1: Level-1 interrupt 0: Other Pipeline Replay (i.e. excludes cache miss etc.)
6	Hold and Other Bubble cycles	15~9: reserved 8: WAITI bubble i.e. a cycle spent in WaitI power down mode. 7: CTI bubble (e.g. branch delay slot) 6: reserved 5: R hold caused by MEMW, EXTW or EXCW 4: R hold caused by register dependency 3: R hold caused by Store release 2: R hold caused by DCache miss 1: reserved 0: Processor domain PSO bubble
7	Instruction TLB Accesses (per instruction retiring)	15~4: reserved 3: ITLB Miss Exception 2: HW-assisted TLB Refill completes 1: Replay of instruction due to ITLB miss 0: ITLB Hit

Table 9–54. Select and Mask Meanings (continued)

Select Field	Description	Mask Bits (Sub-Events to Count)
8	Instruction Memory Accesses (per instruction retiring)	15~4: reserved 3: Bypass (i.e. uncached) fetch 2: All InstRAM or InstROM accesses 1: Instruction Cache Miss 0: Instruction Cache Hit
9	Data TLB Accesses	15~4: reserved 3: DTLB Miss Exception 2: HW-assisted TLB Refill completes 1: Replay of load/store due to DTLB miss 0: DTLB Hit
10 (LS0)[1] 13 (LS1) 16 (LS2)	Data Memory Load Instruction	15~4: reserved 3: Bypass (i.e. uncached) load 2: Load from local memory i.e. DataRAM, DataROM, InstRAM, InstROM 1: Data Cache Miss 0: Data Cache Hit
11 (LS0)[1] 14 (LS1) 17 (LS2)	Data Memory Store Instruction	15~4: reserved 3: PIF Store 2: Store to local memory i.e. DataRAM, InstRAM 1: Data Cache Miss 0: Data Cache Hit
12 (LS0)[1] 15 (LS1) 18 (LS2)	Data Memory Accesses (Load, Store, S32C1I, etc instructions)	15~1: reserved 0: Cache Miss
22	Multiple Load/Store	15~6: reserved 5: 2 stores and 0 loads 4: 0 stores and 2 loads 3: 1 stores and 1 loads 2: 1 stores and 0 loads 1: 0 stores and 1 loads 0: 0 stores and 0 loads
23	Outbound PIF	15~2: reserved 1: Prefetch 0: Castout
24	Inbound PIF	15~2: reserved 1: Data DMA 0: Instruction DMA

Table 9–54. Select and Mask Meanings (continued)

Select Field	Description	Mask Bits (Sub-Events to Count)
26	Prefetch	15~6: reserved 5: Direct fill to (L1) Data Cache 4: reserved 3 D prefetch-buffer-lookup miss 2: I prefetch-buffer-lookup miss 1: D prefetch-buffer-lookup hit 0: I prefetch-buffer-lookup hit
27	iDMA	15~1: reserved 0: active cycles
28	Instruction Length	15: reserved 14: 128-bit 13: 120-bit 12: 112-bit 11: 104-bit 10: 96-bit 9: 88-bit 8: 80-bit 7: 72-bit 6: 64-bit 5: 56-bit 4: 48-bit 3: 40-bit 2: 32-bit 1: 24-bit 0: 16-bit

1. Unless a processor is configured with multiple load/store units, only the select value marked LS0 is used. Specifically, select values 10 through 12 are used for load/store unit 0; select values 13 through 15 for load/store unit 1; and so on.

A counter will only increment by one even if more than one condition corresponding to a set mask bit occurs.

9.5.1 Configurability

The counted events shown in Table 9–54 are subject to configuration restrictions. For example, in configurations with:

- No ICache or DCache, configuring a counter to count only DCache miss or ICache miss (select = 3, mask bit 3)
- A single load/store unit, configuring a counter to count LS1 cache miss or hit (select = 14, mask bits 0 and 1 = 0x0003)

In these cases, the count value will remain unchanged, and `PerfMonInt` is never asserted.

9.5.2 Illogical Use of Counters

In general, there are no hardware additions to catch or ensure that software does not do something misleading. For example, consider a specific case where a Performance Monitor is configured to count store buffer conflict stalls (select = 3, mask bit 2); and on an instruction which is supposed to stall for perhaps 15 cycles, at the 7th cycle of stall, from the APB interface, the control register is changed to count DTLB hits (select = 9, mask bit 0). The performance counter will behave as programmed; in this case it will count up to 7 and then simply stop, or if there are subsequent DTLB hits, count these from 8 onwards.

9.5.3 Accuracy of Counts

The accuracy of Performance Monitor counts depends on the accuracy of the indicated Traceport events.

For events related to the completion of a valid instruction, such as LS0 Data Cache Hits (select = 10, mask bit 0), there is a guaranteed cycle available for the Xtensa core to output the appropriate indication on the Traceport. The information related to the instruction, such as PC, is also unambiguously available. For this reason, we guarantee 100% accuracy in counting such events.

For events not related to the completion of a valid instruction, such as LS0 Data Cache Misses (select = 10, mask bit 1), there may not be a guaranteed cycle available to output the appropriate indication on the Traceport. This is due to the speculative nature of the processor pipeline and the fact that multiple events could be happening concurrently. Moreover, the information associated with the event, such as PC, may not be unambiguous. While it is high, we say that accuracy in counting such events is not 100%.

9.6 Event Details

This section provides more details about what is being counted for the different Select and Mask values of the control registers. The information is not always directly available; in many cases, the Performance Monitor needs logic to decode the event from what is being presented on the Traceport. This is referred to as the event-generating logic.

A note on terminology: “Retiring” (also known as “committing”) refers to an instruction reaching the W stage without being killed. Once an instruction reaches the W stage, its effects on architectural state cannot be reversed. Depending on the instruction, it may continue to update both architectural (e.g., long-latency user-TIE instructions) and microarchitectural states (e.g. DCache refill) beyond the W stage. In general, the performance monitor is unable to count events associated with post-W-stage effects.

9.6.1 Overflow of Lower Counter

As Select value 1 in Table 9–54 shows, one of the events selectable by counter *n* is the overflow of counter *n*-1. This means that when the user sets up counter *n-1* with a given select value and specifies select 1 for the next counter i.e. counter *n*, counter *n* is used to count the overflows of counter *n*-1. This allows concatenation of the two counters to count the performance events selected in the first counter. In this way it is possible to count more than 2^{32} occurrences of a given event.

Counts wider than even 2^{64} are possible by setting more than one adjacent counter with select = 1.

When counter 0 has select = 1, it counts the overflows of the largest numbered counter of the configuration. In other words, modulo arithmetic is used in determining *n*-1.

9.6.2 Retired Instructions

With select = 2, it is possible to get a dynamic count of successfully completing instructions. If all the masks are clear, the counter is off. If all masks are set, the counter counts ALL instructions retired.

If the masks are appropriately set, it is possible to count many types of instructions, such as the static number of unconditional loops (mask bit 8), dynamic number of loop iterations (mask bit 10), fall-through branches (mask bit 7), taken branches (mask bit 4), indexed jump/call (mask bits 5 and 6), returns (mask bit 2), etc.

Notice that the last instruction of a loop (aka. loopback taken and loopback fallthrough) masks are special in that they can happen on an ordinary (i.e., non control-transfer instruction). In these cases, there is no non-CTI (control transfer instruction) count; that is, they would not be counted if only mask bit 15 were set.

9.6.3 Data-related GlobalStall Cycles

A stall cycle happens when `GlobalStall` is asserted. It is indicated on the Traceport as an encoding of `PDebugStatus` when no instruction is retiring in the W stage. The stall indication takes precedence over the indication of other bubble causes, such as register dependency or replay, were these bubbles to happen at the same time. Once the stall goes away, the bubble indication naturally follows.

Stall cycles cannot always be pinned to a single cause alone since different sources of stalls can occur at the same time. For example, an instruction cache miss and a data cache miss can occur at the same time. As a result, it is important to note that the sum of each category counted individually will be more than the stall count of all categories counted at once, with the difference attributable to multiple-stall-condition cycles.

The underlying causes of the `GlobalStall` assertion are given on `PDebugData` as a bit map. In other words, when the given `PDebugData` bit is asserted, the stall cycle corresponding to that mask bit is counted. Refer to the description of the stall in Section 15.3.11.

The data-related causes are as follows:

- Bank-conflict stall (select = 3, mask bit 8) derived from `PDebugData[16]`
- Bypass load stall (select = 3, mask bit 7) derived from `PDebugData[15]`
- Load/store miss-processing stall (select = 3, mask bit 6) derived from `PDebugData[14]`
- Data inbound-PIF request stall (select = 3, mask bit 5) derived from `PDebugData[5]`
- Data RAM/ROM/XLMI busy stall (select = 3, mask bit 4) derived from `PDebugData[4]`
- DCache-miss stall (select = 3, mask bit 3) derived from `PDebugData[3]`
- Store buffer conflict stall (select = 3, mask bit 2) derived from `PDebugData[2]`
- Store buffer full stall (select = 3, mask bit 1) derived from `PDebugData[1]`

Causes grouped under instruction-related `GlobalStall` cycles are:

- Iterative divide stall (select = 4, mask bit 8) derived from `PDebugData[19]`
- Iterative multiply stall (select = 4, mask bit 7) derived from `PDebugData[18]`
- FastL32R stall (select = 4, mask bit 6) derived from `PDebugData[13]`
- Bypass I fetch stall (select = 4, mask bit 5) derived from `PDebugData[12]`
- RunStall (select = 4, mask bit 4) derived from `PDebugData[10]`
- TIE port stall (select = 4, mask bit 3) derived from `PDebugData[9]`
- Instruction RAM inbound-PIF request stall (select = 4, mask bit 2) derived from `PDebugData[8]`
- Instruction RAM/ROM busy stall (select = 4, mask bit 1) derived from `PDebugData[7]`
- ICache-miss stall (select = 4, mask bit 0) derived from `PDebugData[6]`

9.6.4 Exceptions and Replays

With select = 5, it is possible to count interrupts, exceptions, and replays, as follows:

- Mask bit 8 counts hardware-corrected memory errors in configurations that have ECC. Note that these do not result in as exception, i.e., no jump to a vector. The errors are corrected on the fly.
- Mask bit 7 counts exceptions other than those corresponding to bits 1 to 6. Examples of this include memory errors (which again is separate from the hardware-corrected memory errors of mask bit 8), bus errors, illegal instructions, syscall, instruction-fetch-related errors, load/store-related errors, TLB-related errors, coprocessor disabled, double exceptions, etc.
- Mask bit 6 corresponds to an Allocate exception that has EXCCAUSE set to 5. See the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for more details.
- Mask bit 5 corresponds to a window overflow or underflow exception.
- Mask bit 4 corresponds to an NMI.
- Mask bit 3 corresponds to a Debug exception.
- Mask bit 2 corresponds to greater-than-level-1 interrupts.
- Mask bit 1 corresponds to a level-1 interrupt that has EXCCAUSE set to 4. See the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for more details.

- Mask bit 0 corresponds to replays. A replay could happen for a number of reasons. In many cases, replays happen in the same cycle that other events are signaled on the Traceport. Specifically, these are:
  - ICache or DCache miss
  - TLB miss
  - exception or interrupt
  - Hardware-assisted TLB refill
  - Hardware-corrected memory error

In other words, when counting replays using select = 5 and mask bit 0, it is important to remember that only replays that happen at cycles not coinciding with the above events are counted. The events in the list above can be counted using other select/mask values, and always result in replays.

9.6.5 Hold and Other Bubble Cycles

In this section, we refer to “bubbles” and “holds”. To clarify, a bubble occurs when there is no valid instruction retiring in a cycle. A bubble is the combination of all possible reasons that there is no valid instruction in the W-stage. A hold, one of the possible reasons for a bubble, happens when an instruction is prevented from moving past the R stage due to a pipeline conflict. Other causes of bubbles include branching or pipeline flushing.

Hold and other bubble counting is based on the Traceport reporting of these events, that is, register dependency, resource conflict, and control transfer. Bubbles are reported as encodings of `PDebugStatus` when `PDebugInst` indicates that there is no valid instruction retiring. Being a subcategory of bubbles, holds are reported on `PDebugData` as a bit map, and when `PDebugInst` indicates that there is no valid instruction retiring, and when `PDebugStatus` has a specific encoding (0000_10) that says that the bubble is because of a hold. Bubble and hold cycles that are countable are as follows:

- Bubble due to PWait mode (select = 6, mask bit 8) derived from `PDebugStatus`=1010_00. This is a cycle where the core is in low-power mode due to WAITI.
- Bubble due to a CTI instruction (select = 6, mask bit 7) derived from `PDebugStatus`=0001_00.
- MEMW, EXTW or EXCW instruction dependency (select = 6, mask bit 5) derived from `PDebugData[16]` when `PDebugStatus` = 0000_10. Upon executing a barrier instruction (i.e. MEMW, EXTW or EXCW) the pipeline is held to wait for prior instructions e.g. stores to complete.

- Register dependencies or resource conflicts (select = 6, mask bit 4) derived from PDebugData[12] when PDebugStatus = 0000_10. These holds are bubbles due to register dependencies between instructions or conflicts for resources such as TIE ports.
- Store release dependency (select = 6, mask bit 3) derived from `PDebugData[11]` when `PDebugStatus` = 0000_10. The pipeline is held to allow synchronization semantics to complete.
- Holds or bubbles that wait for various LoadStore dependencies e.g. miss handling, castouts, prefetch, cache access insts, s32c1i, etc. to be resolved (select = 6, mask bit 2) derived from `PDebugData[8]` when `PDebugStatus` = 0000_10.
- Bubble due to processor domain being shut off (select = 6, mask bit 0) derived from PDebugStatus=0000_00. These are cycles that the processor spends in the power shut-off state. This can be meaningfully counted only when the processor domain is powered down and the debug power domain is powered up.

9.6.6 Instruction TLB Accesses

The ITLB-related events are:

- Miss-Exception event: Instruction TLB miss-exception because there was an ITLB miss, and the hardware ITLB refill operation itself then took a TLB-Miss exception. This event is created by looking at the EXCCAUSE available on `PDebugData`.
- HW-TLB refill event: An ITLB miss occurred, and a hardware TLB refill operation successfully completed. This event is created when `PDebugStatus` = 0110_00.
- Replay of an instruction due to ITLB miss.
- Hit event: An instruction is retiring and it used the instruction TLB without a miss. Indicated by `PDebugStatus` = 0001_10.

In the process of executing an instruction, more than one of the above events might be indicated. That is, an instruction that gets an Instruction HW-TLB refill will subsequently get an Instruction TLB hit (immediately if not killed upon the replay for an unrelated reason). Similarly, any instruction that gets an Instruction TLB Miss-Exception, will subsequently get an Instruction HW-TLB refill, and after the Instruction HW-TLB refill, get an Instruction TLB hit. So, the sum of separately-counted events will be more than the per-instruction ITLB access count.

9.6.7 Instruction Memory Accesses

Following are the instruction fetch events:

- Cache Miss: An instruction cache miss occurred, and a request was sent to the PIF.
- Cache Hit: An instruction in the W-stage is valid and is executing out of the cache.
- InstRAM: An instruction in the W-stage is valid and is executing out of one of the InstRAMs or InstROM or L0 buffers.
- Uncached: Fetch of an instruction from a bypass space.

Instruction cache miss is processed early in the pipeline, and is counted when the missing instruction would have been in the W-stage. An ICache miss will be counted once as a miss and once as a hit when the fill completes and the instruction successfully executes. When fetch is happening from bypass-attribute memory and the instruction is unaligned, two bypass fetches may be required before the instruction can execute.

9.6.8 Data TLB Accesses

Following are the DTLB-related events:

- Miss-Exception: Data TLB miss-exception because there was a DTLB miss, and the hardware TLB refill operation itself then took a TLB-Miss exception. This event is created by looking at the EXCCAUSE available on `PDebugData`.
- HW-TLB refill: A DTLB miss occurred, and a hardware TLB refill operation successfully completed. This event is created when `PDebugStatus` = 0111_00.
- Replay of a load/store due to a DTLB miss.
- Hit: Any load/store that is retiring in the W-stage and that used the data TLB without a miss indication on `PDebugLS`*n*`Stat[7]`.

More than one of the above events might be indicated in the process of performing a load or store. That is, a load/store that gets a Data HW-TLB refill will subsequently get a Data TLB hit (immediately, if not killed upon the replay for an unrelated reason). Similarly, any load/store that gets a Data TLB Miss-Exception will subsequently (usually many instructions later) get a Data HW-TLB refill, and after the HW-TLB refill, get a Data TLB hit.

9.6.9 Loads

Following are the load events:

- Cache Load: There is a retiring load instruction in the W-stage that is using the cache.
- Load Miss: There is a load in the W-stage that is meant to use the cache but is going to get data over the PIF.

- DataRAM Load: There is a retiring load instruction in the W-stage that is from local RAM – i.e. DataRAM or DataROM or XLMI.
- Load from regions with attribute “Bypass” or “No Allocate”.

9.6.10 Stores

Following are the store events:

- Cached Store: There is a retiring store instruction in the W-stage that is using the cache.
- Store Miss: There is a load in the W-stage that is meant to use the cache but is going to get data over the PIF. In write-back configurations, this may also entail a cache refill.
- DataRAM Store: There is a retiring store instruction in the W-stage that is to local RAM—i.e. DataRAM, DataROM, or XLMI.
- PIF Store: There is a retiring store in the W-stage that will be sent to the PIF (write-through, bypass, or no-allocate attribute). It may also be sent to the cache if the data is in the cache (write-through hit, no allocate hit).

S32C1I instructions are counted as part of the store categories. They are counted simply when/if they retire, regardless of the result of the compare, that is, without regard to whether the write to the target memory happened.

9.6.11 Data Memory Accesses

These are events related to any load/store/S32C1I instructions.

- Cache Miss: Load/store in W that was supposed to hit in the cache but did not.

9.6.12 Multiple Load/Store

With select = 22, it is possible to count multiple load/stores.

9.6.13 Outbound PIF Transactions

The outbound PIF events are indicated on the PDebugOutPIF Traceport signal.

- Castouts: This event occurs when the Xtensa core begins a block write on the outbound PIF.
- Prefetches: This event occurs when the Xtensa core begins a prefetch read block on the outbound PIF.

9.6.14 Inbound PIF Transactions

The inbound PIF events are indicated on the `PDebugInbPIF` Traceport signal. The type of events counted are as follows.

- Error: Address does not fall in a legitimate target
- Data: A data RAM was the target
- Instruction: An instruction RAM was the target

The event is signaled on the Traceport (therefore available to the Performance Monitor) when the response data valid signal is asserted the first time.

9.6.15 Prefetch

There are three basic metrics that may be used to measure prefetch effectiveness as follows:

1. (select = 26, mask bit 0 to 3) The number of prefetch-buffer-lookups of the prefetch buffers generated by the pipeline in response to a cache miss (or due to a software-directed prefetch instruction). Following are four sub-categories:
   a. (select = 26, mask bit 0) a prefetch-buffer-lookup, for an instruction fetch, that hit in the prefetch buffers: this indicates that prefetch processing for an instruction fetch has already been initiated
   b. (select = 26, mask bit 1) a prefetch-buffer-lookup, for a data access, that hit in the prefetch buffers: this indicates that prefetch processing for a load/store miss has already been initiated
   c. (select = 26,mask bit 2) a prefetch-buffer-lookup, for an instruction fetch, that misses in the prefetch buffers: this means that the instruction cache line is not available in the prefetch buffer, and will be fetched by a regular ICache miss request to the PIF
   d. (select = 26,mask bit 3) a prefetch-buffer-lookup, for a data access, that misses in the prefetch buffers: this means that the data cache line is not available in the prefetch buffer, and will be fetched by a regular DCache miss request to the PIF
2. (select = 23, mask bit 1) The number of prefetch-related PIF block reads.
3. (select = 26, mask bit 4) The number of fills to the L1 cache from the prefetch buffers. Tying this back to the first metric, fills to the L1 obviously reduce the number of prefetch-buffer-lookups to begin with. Note that not all prefetches are destined to L1 (inst pref, DL1 clear, etc.)

9.7 Profiling Code with the Performance Monitor

The Performance Monitor allows the user to profile code. However, to get certain code-profiling metrics, especially those that are similar to the metrics provided by the ISS, it might be necessary to collect and combine multiple performance counts. An example where different performance counts need to be combined to get the full overhead cycle count is data cache miss penalty. Here we need to combine the additional cycles due to data cache stalls (select = 3, mask bit 3 and mask bit 6) with the data cache hold penalties (select = 6, mask bit 2). Also we need to add the replay penalties, which is number of pipeline stages of the core times the data cache miss ratio (select = 10 or 13 depending on single or dual load/store, mask bit1).

Since it is necessary to use a recipe that combines various counts appropriately, the number of counters selected at processor-configure time must be considered carefully.

Following are recipes for some metrics of common interest.

To reiterate, to obtain a desired performance metric, it may be necessary to configure and simultaneously use multiple performance counters. For example, data cache miss penalty would require either the simultaneous use of three counters, or running the profiling code sequentially three times with different select and mask settings on the single counter.

9.7.1 Retired Instructions

To count all committed instructions reaching the W-stage, use select = 2 and set all mask bits i.e. a mask value of 0xffffffff.

9.7.2 Branch Penalty

Branch penalties may be counted using select = 6 and mask bit 3 i.e. a mask value of 0x0080.

9.7.3 Pipeline Interlocks

The number of interlocks, meaning R-stage hold due to register dependencies and interlock-specific instructions, may be counted using select = 6 and mask bits 3,4 and 5 i.e. a mask value of 0x0038. If register dependencies alone are of interest, use mask value 0x0010. If iterative multiply and divide bubbles are to be counted as part of “pipeline interlocks”, add select = 4 and mask value 0x0180.

9.7.4 Instruction Cache Miss Penalty

To count instruction cache miss penalty, meaning bubble cycles waiting for an instruction fetch, use the following equation:

ICache_miss_penalty = Num_ICache_stalls + Num_ICache_misses * Pipeline_penalty

- Num_ICache_stalls may be counted using select = 4 and mask bit 0 i.e. value of 0x0001.
- Num_ICache_misses may be counted using select = 8 and mask bit 2 i.e. value of 0x0002.
- Pipeline_penalty is configuration-dependent. It is 3 for a 5-stage pipeline or 4 for a 7-stage pipeline.

9.7.5 Data Cache Miss Penalty

To count data cache miss penalty, meaning bubble cycles waiting for load or store miss to be processed, use the following equation:

DCache_miss_penalty = Num_DCache_stalls + Num_DCache_holds + Num_D-Cache_misses * Pipeline_depth

- Num_DCache_stalls may be counted using select = 3 and mask bits 6 and 3 i.e. value of 0x0048 and subtracting a count with mask bit 7 i.e. value 0f 0x0080.
- Num_DCache_holds may be counted using select = 6 and mask bit 2 i.e. value of 0x0004.
- Num_DCache_misses may be counted using select = 10 (or 13 or both as appropriate) and mask bit 1 i.e. value of 0x0002.
- Pipeline_depth is simply the length of the pipeline of the configuration i.e. 5 for 5-stage pipeline or 7 for 7-stage pipeline.

As previously mentioned, this requires the simultaneous use of multiple counters, or with one counter requires the rerunning of profiling code with different counter settings. If bypass (uncached) accesses, DCache special instructions, or block prefetch instructions are a significant number compared to DCache misses, then the DCache Miss Penalty equation will not be accurate because the hold cycles for those items are included in the Num_Dcache_holds count.

9.8 Software

The previous sections described the hardware implemented for performance monitoring. The Cadence-provided software tools and methods to analyze a program run using the hardware monitors is described in the Hardware-Based Performance Analysis chapter of the *Xtensa Software Development Toolkit User's Guide*.

10. TRAX Overview

10.1 Introduction

TRAX is a collection of hardware and software that provides increased visibility into the activity of running Xtensa processors using compressed execution traces. TRAX is particularly useful for software and system development and debugging.

TRAX provides program execution trace — but no data trace. Using a circular Trace-RAM and a simple set of triggers, the processor's flow of execution around a trigger point of interest is compressed in realtime and captured for later analysis. A software tool (`xt-traxview`) converts captured traces into an annotated sequence of assembly instructions.

Note: TRAX requires version 7.1.0 or later of the Xtensa OCD Daemon.

A working TRAX environment consists of various software and hardware components. Figure 10–30 illustrates these components for a single TRAX unit system. Note that the TRAX compressor is not a standalone module but a part of the Debug module.

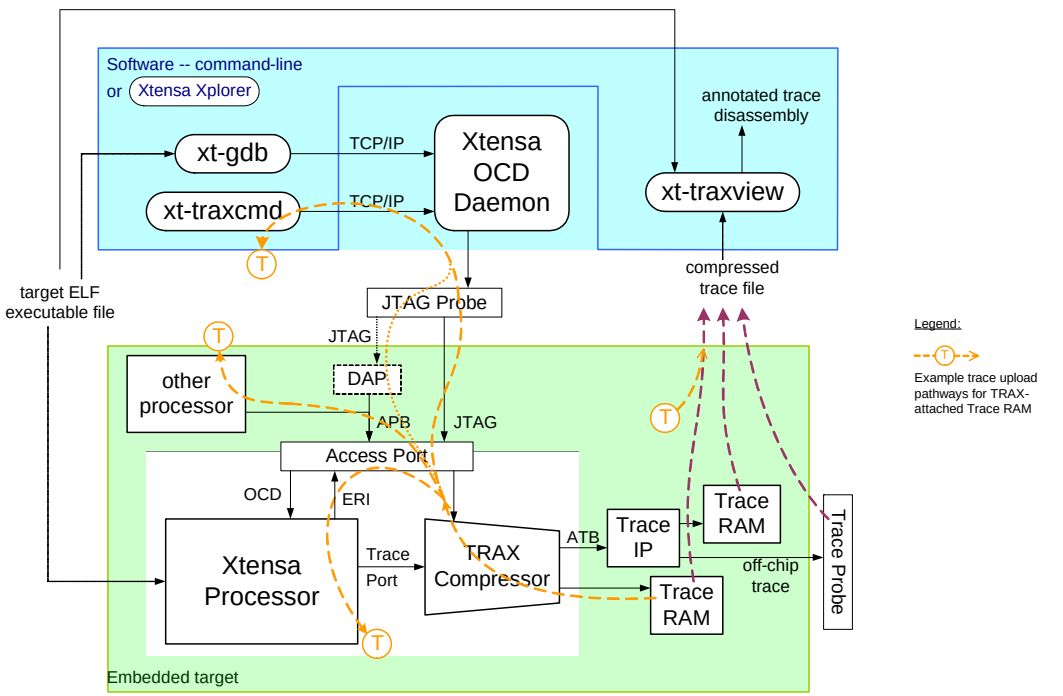


Figure 10–30. TRAX Environment

The TRAX compressor monitors the processor's trace port and records changes in the processor's flow of execution into a circular or external TraceRAM. The format of the trace information in the TraceRAM is very similar to that of the Nexus 5001 standard, with minor variations as detailed in Section 15.3. This information consists of compressed descriptions of taken branches, exceptions, and interrupts. It is sufficient to reconstruct the processor's flow of execution, including the value of the program counter (PC) for committed instructions.

The `xt-traxcmd` tool (Chapter 11) uses the Xtensa OCD Daemon (Chapter 6) to manage the TRAX compressor under user-control and capture resulting traces. Alternatively, the TRAX library (Chapter 13) can be used an Xtensa processor's trace itself or from another processor. However the trace file is obtained, the `xt-traxview` tool (Chapter 12) makes these traces human-readable.

10.2 Hardware Structure

The TRAX compressor is also referred to as the trace compressor or the "TraceCompressor". It is important to note here that while "compressor" is convenient shorthand, the TraceCompressor also performs other functions. For example, it houses all the control and status registers used in programming TRAX, incorporates trigger logic (including PC trigger), performs TraceRAM control, and so forth. TraceCompressor is the central trace generation and control logic of TRAX, and it is part of the Debug module.

There are multiple ways to control and access TRAX. The traditional way is through the JTAG interface to Xtensa. If APB access is configured, Xtensa will have an APB slave port which provides access to the TRAX registers. Finally, through the ERI (External Register Interface), the processor core has access to its own TRAX. As Figure 10–30 shows, these multiple interfaces go through the Access Port. The Access Port is a submodule of Xtensa that implements the interface logic. For example, it has the TAP FSM required by JTAG.

The TRAX registers appear as a simple memory-mapped space to all interfaces. This is possible for JTAG also because the Access Port uses the IEEE-ISTO Nexus 5001 standard to implement the JTAG access.

10.3 TRAX Features

- Works with Xtensa LX7 and Xtensa NX1 processors, and Diamond Standard Cores
- Traces all changes in the program flow, including exceptions and interrupts
- All functionality is accessible over the processor's standard JTAG connection
- Approximately 1000 to 2000 instructions traced per KB of TraceRAM (typical)

- Configurable circular TraceRAM size from 512 bytes to 256 KB
- Support to capture traces before, around, or after the trigger, or some distance after the trigger
- Correlates trigger occurrence with traced instruction stream
- Selectable trigger sources, including one program counter trigger with power-of-2 range, two external trigger inputs (processor break-in / break-out and trace trigger network), and manual triggering
- Two selectable external trigger outputs (processor break-in / break-out and trace trigger network)
- Unobtrusive — does not affect processor execution timing[6]
- Fully supported by Xtensa Xplorer IDE and with scriptable command-line utilities
- TRAX is part of the Debug module (see Chapter 1)
- Accessible over APB, and from core itself (ERI), in addition to JTAG
- Support for external tracing over an ATB (AMBA Trace Bus) interface
- Shareable local TraceRAM, using boundary registers and RAM busy signals

10.4 Processor Configuration Requirements

TRAX requires the following processor configuration options to be enabled: the trace port, On-Chip-Debugging (OCD), and an external debug interrupt (also referred to as *break-in* / *break-out*).

Diamond Standard cores are preconfigured with these options.

10.5 TRAX Details

TRAX details are described in the following chapters:

- Chapter 11 describes the `xt-traxcmd` tool that provides external access to TRAX.
- Chapter 12 describes the `xt-traxview` tool that decompresses a TRAX trace file.
- Chapter 13 describes the TRAX library that provides embedded access to TRAX.
- Chapter 6 describes how to set up the Xtensa OCD Daemon to use TRAX.
- TRAX hardware is described in Chapter 14, including the details on how to incorporate TRAX hardware in an SoC or FPGA.

6. While the timing of the running processor itself is not affected, the timing of on-chip debugging (OCD) operations over a shared JTAG chain can be affected. However, the effect of such timing variances is negligible. Given their timescale, OCD operations are generally not timing sensitive.

- Chapter 18 describes the `xt-trace` tool that decodes raw Xtensa trace port data. Since this tool can be used both in the simulation environment or with actual hardware, this chapter is useful for both hardware and software engineers.
- Chapter 15 describes the programmer's model of the TRAX compressor. This chapter contains material that is relevant to trace and debug tool developers. Some of this information may also be of interest to TRAX users.

For more details on the use of TRAX within Xtensa Xplorer, refer to Xtensa Xplorer online Help. For details on the Xtensa Traceport, see Chapter 16.

11. TRAX Control Tool (xt-traxcmd)

The TRAX compressor hardware is controlled by the xt-traxcmd command-line tool, which connects to the Xtensa OCD Daemon over TCP/IP, or by a target resident TRAX library (see Chapter 13). The Xtensa OCD Daemon must be set up as described in Chapter 6 and be connected to a live target before invoking xt-traxcmd.

Note: xt-traxcmd version 10 is incompatible with the Xtensa OCD Daemon version 9 or older.

11.1 Synopsis

The xt-traxcmd tool accepts the following command-line options:

```
xt-traxcmd [-h|--host hostname]
           [-p|--port portnum]
           [-c|--commandfile filename]
           [-t|--tracefile filename]
           [-u|--unit devno]
           [--mi]
           [--ocd-disabled]
```

xt-traxcmd displays a brief usage summary when you use the --help option.

11.2 Command-Line Arguments

This xt-traxcmd tool understands a simple set of commands that allow you to set up the TRAX compressor to capture processor execution traces and download the resulting trace data. These commands can be sourced from a file, allowing common invocations of this tool to be scripted.

11.2.1 Xtensa OCD Daemon Host Name (--host hostname)

This option specifies the DNS host name or IP address (in a *xx.xx.xx.xx* form) of the Xtensa OCD Daemon.

This parameter defaults to localhost, which generally refers to the local machine.

11.2.2 Xtensa OCD Daemon Port Number (--port portnum)

This option specifies the Xtensa OCD Daemon's TCP port number for controlling TRAX. This number must match the TRAX application port number specified in the topology file (as described in Section 6.7). The default, if unspecified, is 11444.

This option is usually not needed unless multiple Xtensa OCD Daemons are running on the same machine.

11.2.3 Command File (--commandfile file)

This option specifies a text file from which to read commands defined in Section 11.3 "Command Language". Once the end of the file is reached, or if any error is encountered or the `quit` command executed, `xt-traxcmd` exits.

The format of the command file is the same as those executed using the `source` command (Section 11.3.10). Empty lines and any part of a line starting with the hash character (#) are ignored as comments. Any non-zero amount of white space (spaces and tabs) may be used to separate commands and arguments.

If this option is not specified, `xt-traxcmd` enters interactive mode where it prompts for commands until the `quit` command is entered.

11.2.4 Trace File (--tracefile file)

This option specifies a default file to which captured trace data is saved. All invocations of the `save` command (see Section 11.3.6 on page 211) overwrite trace data to this file. The filename specified is not tested for validity until the `save` command is invoked.

If this option is not specified, the `save` command reports an error if not given an explicit trace file name.

The format of the generated trace file is described in Section 15.4.

11.2.5 TRAX Device Selection (--unit devno)

This option specifies which TRAX device to select initially. The `select` command (see Section 11.3.7) can be used to change the selection during the `xt-traxcmd` session.

If this option is not specified, TRAX device zero (0) is selected initially.

11.2.6 Machine Interface Mode (--mi)

This option enables a machine interface mode that simplifies controlling `xt-traxcmd` from another program. Commands and output are formatted in a manner similar to that supported by the GDB `--mi` option. This option is used by Xtensa Xplorer.

11.2.7 OCD Selection (--ocd-disabled)

This option is only relevant for TRAX version 1.0, which is supported with Diamond RevA processors only. It is ignored for TRAX versions 1.1 and later, or in other words, with Xtensa LX2 or Xtensa X7 or Diamond RevB and later processors.

This option indicates whether a debugger is attached to the processor using OCD during the trace capture session. By default, OCD is assumed to be enabled on the Xtensa processor. Specify the `--ocd-disabled` option if OCD is disabled. That is, specify this option when the processor is not being debugged via OCD while tracing.

When OCD is disabled, debug exceptions are handled by the processor's debug exception vector rather than by stopping. Thus, when GDB attaches to the processor using a mechanism other than OCD (such as using a serial port talking to a debug agent on the target), the `--ocd-disabled` option must be specified.

`xt-traxcmd` simply stores this information in all saved trace capture files. `xt-traxview` uses the information to properly decode any debug exceptions present in the captured trace stream.

11.3 Command Language

The `xt-traxcmd` tool understands the following set of commands (the command names are case-insensitive):

- `halt`
- `help [set | show]`
- `poll`
- `quit`
- `reset`
- `save [filename]`
- `select devno`
- `set parameter value`
- `show [parameter]`
- `source filename`

- start
- status
- stop
- version
- wait

Each of these commands are described in the following subsections.

Numeric parameters can be decimal, octal (prefixed with 0) or hex (prefixed with 0x). There is no support for expression evaluation or control flow.

11.3.1 Halt Tracing Immediately (halt)

This command stops all trace capturing immediately, including any selected post-stop-trigger capture. If trace capture is not in progress, a warning is reported.

Note for TRAX 1.0 only: Due to an errata in TRAX 1.0 hardware, the halt command cannot complete if the processor remained completely idle since the trace start command. This can happen, for example, when executing start and halt (or start and stop) commands in succession while the processor is stopped under control of the debugger using OCD. As a workaround, have the debugger cause the processor to execute at least one instruction before executing the xt-traxcmd halt (or stop) command. This may be done by querying a processor register (if using xt-gdb), by viewing a new area of target memory, or by single-stepping the processor. This issue does not exist in TRAX 1.1 and later hardware.

11.3.2 Display Available Commands (help [set |show])

The help command displays a list of available commands.

Use set or show as an argument to display the available parameters for set and show commands.

11.3.3 Poll Status (poll)

The poll command displays a brief machine readable status.

11.3.4 Exit xt-traxcmd (quit)

This command ends the xt-traxcmd session. This happens regardless of whether the command is invoked in a command file (specified per Section 11.2.3), in a command file read with the source command, or entered interactively.

Note: Command files read with the `source` command complete normally and return by reaching the end of the file. No special command is needed to return. Invoking the `quit` command at the end of a command file exits `xt-traxcmd`, not the `source` command.

11.3.5 Reset TRAX (`reset`)

This command resets TRAX parameters as described in Table 11–55 on page 213. This corresponds to resetting TRAX registers.

Note: This command currently avoids a true hardware reset. It simply initializes each TRAX register. TRAX implementations can only be hardware-reset by asserting `DReset`. This command is also not a JTAG or TAP reset which is intrusive, affecting all other devices on the same JTAG scan chain.

11.3.6 Save Captured Trace to a File (`save [filename]`)

This command reads the most recently captured trace still present in the TraceRAM, if any, and writes it to a trace file. If a filename is not specified, the trace filename specified using the `--tracefile` option (see Section 11.2.4) is used.

If trace is active, an error is reported instead. The TraceRAM cannot be accessed while trace capture is in progress. In a command file, for example, this requirement can be met by executing a `wait` or `halt` command before the `save` command.

11.3.7 Select Current TRAX Device (`select devno`)

When multiple TRAX devices are present on the scan chain, this command selects the TRAX device being operated on. TRAX devices are numbered contiguously starting from zero according to the order in which they are specified in the topology file.

By default, the TRAX device specified using the `--unit` *`devno`* option is selected. Otherwise, TRAX device number zero is selected.

11.3.8 Set Parameters (`set parameter value`)

This command sets the value of the specified TRAX parameter. Accepted *parameter* names and associated valid *value* arguments are listed in Table 11–55 on page 213. See Section 11.4 on page 213 for a full description of each parameter.

11.3.9 Display Value of the Parameter (show [parameter])

This command displays the value of the specified TRAX parameter. Table 11–55 lists accepted *parameter* names. See Section 11.4 for a full description of each parameter.

Invoking `show` without any parameter displays the state of most parameters.

11.3.10 Read Commands from File (source filename)

This command directs `xt-traxcmd` to read commands from the specified text file. After reaching the end of the file, control returns to the command file or prompt that invoked the `source` command.

In this file, empty lines and any part of a line starting with the hash character (#) are ignored as comments. You can use any non-zero amount of white space (spaces and tabs) to separate commands and arguments.

Command files executed with `source` can nest up to 10 levels deep.

Note: When a command file invokes another command file using `source` with a relative pathname, it is currently undefined whether that pathname is relative to the current directory at the time `xt-traxcmd` was invoked, or relative to the directory containing the invoking file.

11.3.11 Start Tracing (start)

This command starts tracing with current parameter settings. If tracing is already in progress, an error is reported. Otherwise tracing starts, and any unsaved contents of the TraceRAM is discarded. This command completes immediately, it does not wait for trace to complete (see `wait` command for that purpose).

11.3.12 Display Current Status (status)

This command provides an informative free-form display of various status information about the TRAX compressor. This information includes whether trace is active, trigger setup and occurrence, TraceRAM size, and other information gleaned from TRAX registers.

11.3.13 Stop Tracing (stop)

This command initiates a stop trigger if one has not already occurred. Any selected post-stop-trigger capture (see `postsize` parameter in Section 11.3.8) proceeds normally.

If trace capture is not in progress, or a stop was already triggered, a warning is reported.

If no post-stop-trigger capture has been selected (postsize parameter is off), the stop command is essentially equivalent to the halt command.

Note: Due to an errata in TRAX 1.0 hardware, the stop command cannot complete if the processor remained completely idle since the start command. See Section 11.3.1 “Halt Tracing Immediately (halt)” for more details.

11.3.14 Display Program Version (version)

This command simply displays the version of xt-traxcmd.

11.3.15 Wait for Trace Capture Completion (wait)

This command waits for trace capture to be complete, including any selected post-stop-trigger capture.

You can interrupt this command using Ctrl-C. Interrupting the command in this way does not cause trace to stop. Use the halt or stop command, as appropriate, for that purpose.

This command completes immediately if trace capture is not in progress.

11.4 TRAX Parameters

Table 11–55 summarizes the TRAX parameters supported by the show and set commands. More detailed descriptions follow.

Table 11–55. Summary of TRAX Parameters

Parameter	Possible Values	Reset Value	Description
bien	off, on	on	Selects break-in pass-through to the processor (PTO) (prior to TRAX 3.0 only)
boen	off, on	on	Selects break-out pass-through from the processor (PTI) (prior to TRAX 3.0 only)
ctistop	off, on	off	Selects stop trigger via cross-trigger input (CTI)
cto	off, on	off	Reports cross-trigger output (CTO) state (show only)
ctowhen	off, ontrig, onhalt	off	Selects condition that asserts cross-trigger output (CTO)
pcstop0	off, [!]*address*[:*size*]	off	Selects stop trigger via PC range match

Table 11–55. Summary of TRAX Parameters (continued)

Parameter	Possible Values	Reset Value	Description
postsize	off, *size*, *size*%, *size*i	off	Determines how much trace to capture past stop trigger
ptistop	off, on	off	Selects stop trigger via processor trigger input (PTI)
pto	off, on	off	Reports processor-trigger output (PTO) state (show only)
ptowhen	off, ontrig, onhalt	off	Selects condition that asserts PTO
syncper	off, on, auto 8, 16, 32, 64, 128, 256	256	Selects the trace synchronization message period
startaddr	address	0	Sets the trace start address in the TraceRAM
endaddr	address	traceRAM_words - 1	Sets the trace end address in the TraceRAM
aten	off,on	off	Enables trace output to be sent to ATB
tmen	off,on	on	Enables trace output to be sent to TraceRAM (in configurations with ATB present)
atid	value	0	If trace output is directed to ATB, indicates the ATB source ID
tsen	off,on	off	Enables timestamps in the TRAX output

The values of TRAX parameters directly reflect the state of TRAX hardware registers. They do not correspond to internal xt-traxcmd state.

11.4.1 Stop on PC Match Parameter (pcstop0)

The pcstop0 parameter specifies an address or address range to match against the processor program counter, or off to disable this feature. Trace stops when the processor executes an instruction matching the specified address or range.

The value of this parameter is in the form:

[**!**]*address*[**:***size*]

If **:***size* is specified, where *size* must be a power of 2, trace stops when the processor executes in the *size* byte aligned range of *size* bytes that contains the specified *address*. Otherwise trace stops if the processor executes the instruction that starts at *address*. The ! prefix inverts the range: if specified, trace stops when the processor executes an instruction that does not match the specified address or range.

11.4.2 Post Stop Trigger Capture Parameter (postsize)

The postsize parameter determines how much trace to capture past the stop trigger. When set to off (the default) or 0 (zero), trace halts immediately upon encountering the stop trigger. Otherwise, postsize can be set to one of the following types of values:

- the number of bytes to trace
- a number suffixed with %, indicating the percentage of the trace memory size (could be the TraceRAM or the size between the start address and the end address, if specified by the user) to trace
- a number suffixed with i, indicating the number of instructions to trace

The maximum post trigger size is 64 MB or 16 million instructions, regardless of the size of the trace size. Due to internal buffering of the TRAX unit, the exact number of bytes or instructions traced past the stop trigger may vary slightly from the amount requested.

Note: Once the stop trigger is hit, TRAX hardware decrements this parameter directly. Hence it must be set again for each new trace started.

11.4.3 TraceRAM Start Address (startaddr) and End Address (endaddr)

The startaddr and endaddr parameters indicate the memory addresses at which the trace is stored inside the TraceRAM. These can be specified by the user and can take any value between 0 to traceRAM_words - 1. The reset value is 0 and traceRAM_words - 1 for the start and the end address parameters, respectively.

11.4.4 ATB Enable (aten) and ATB Source ID (atid)

The output of the trace port can be directed either to the TraceRAM and/or to the ATB. In order to direct the trace to ATB, the aten needs to be enabled. By default, it is set to off. The source ID can be set by the user using the atid parameter. Its default value is zero. The source ID will only be reliably set when the trace is directed to ATB, i.e. aten is on.

11.4.5 Trace Synchronization Period Parameter (syncper)

The syncper parameter selects the trace synchronization period. It is usually best to let xt-traxcmd select the optimal value using the following set command:

```
set syncper auto
```

Possible values of syncper include off, 8, 16, 32, 64, 128, and 256. If syncper is off, no periodic synchronization messages are output. Otherwise, at least one of every *N* trace messages output is a synchronization message.

Setting `syncper` to `on` selects a period from 8 to 256 according to the size of the TraceRAM.

Setting `syncper` to `auto` is equivalent to turning it `off`, because with TRAX 1.1 it is possible to fully synchronize a trace from its terminating synchronization message. This may or may not be true of future implementations.

Note: With the older TRAX 1.0 hardware, traces that capture entry into OCD *Stopped* state are not decodable without proper synchronization. Thus, when using TRAX 1.0 without the `--ocd-disabled` option (see Section 11.2.7), setting `syncper` to `auto` is the same as setting it to `on`.

11.4.6 External Trigger Parameters

The remaining TRAX parameters control cross-triggering between and among Xtensa processors and TRAX units. Figure 11–31 provides a generalized overview of minimal expected connections among these components.

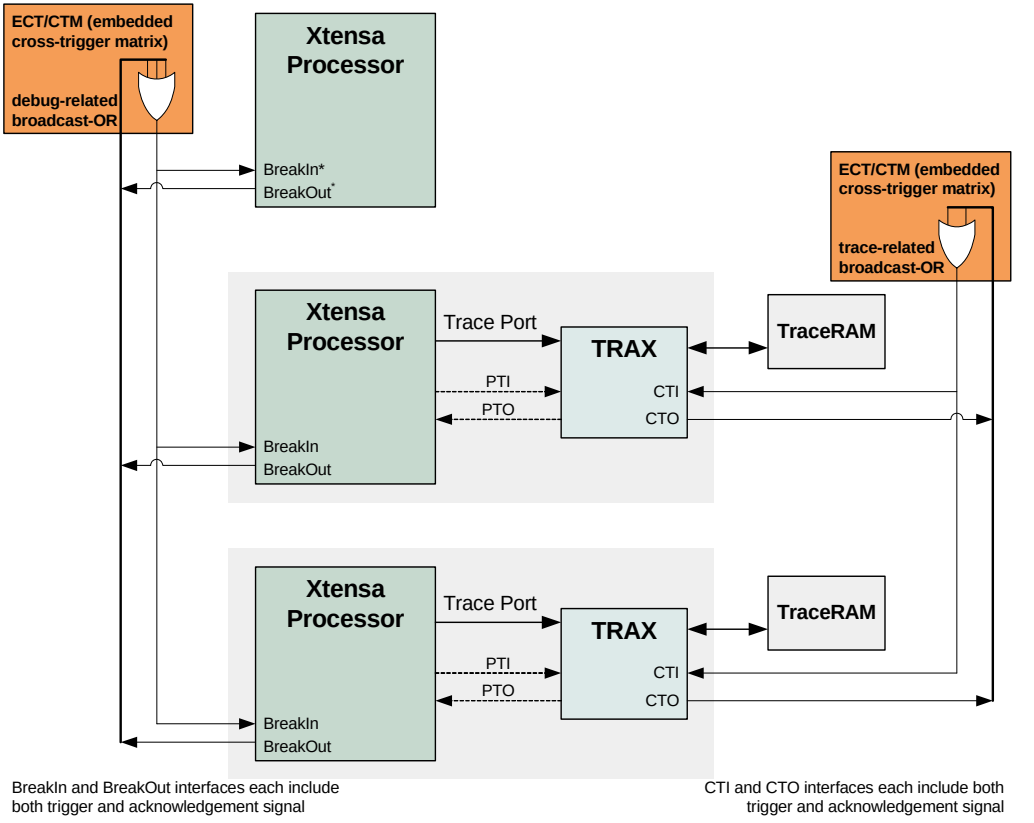


Figure 11–31. TRAX Trigger Interface

Each TRAX unit provides a pair of cross-trigger input and output signals: CTI and CTO. Except in the presence of an external cross-trigger controller, these are typically wired with an OR gate as shown in Figure 11–31, connecting all TRAX units in a system. Each TRAX unit can be configured to assert CTO upon hitting a trace trigger, and/or to trigger upon CTI assertion. Thus, an arbitrary subset of TRAX units can trigger at the same time when any one of them (or any one of some other subset of TRAX units) hits some internal trace trigger.

This cross-trigger network is non obtrusive: it does not affect core execution. It is similar to, but independent from, the debug breakin / breakout network used to synchronously stop cores (into *Stopped* state) on debug events (shown at left in Figure 11–31).

The following parameters control cross-triggering via CTI and CTO:

- The ctowhen parameter selects what internal event, if any, causes assertion of the cross-trigger output (CTO). When set to off, internal triggers have no effect on CTO. When set to ontrig, CTO asserts when the stop trigger fires. When set to onhalt, CTO asserts when all trace capture has halted, including any post stop trigger capture specified with the postsize parameter.
- The ctistop parameter selects whether to trigger trace stop when the cross-trigger input (CTI) signal is asserted. CTI is usually tied to cross-trigger outputs of other TRAX units, or to custom external trigger sources.
- The cto parameter simply reflects the current state of the cross-trigger output (CTO) signal. It cannot be set.

Trace events can trigger debug events and vice-versa. For historical reasons, this is described in terms of PTI and PTO (processor trigger input and output) signals between TRAX and OCD which are now internal to the core.

Note: Except for the pto=onhalt setting, enabling PTO or PTI typically defeats post-trigger tracing.

The following parameters control propagation of trace events to and from debug events:

- The ptowhen parameter allows stopping the processor in response to a trace capture event. It selects what TRAX event, if any, causes a processor debug interrupt via PTO (processor trigger output). When set to off, TRAX triggers have no effect on PTO. When set to ontrig, PTO asserts when the TRAX stop trigger fires. When set to onhalt, PTO asserts when all trace capture has halted, including any post stop trigger capture specified with the postsize parameter.
- The ptistop parameter selects whether to stop tracing (trigger trace stop) when the processor takes a debug exception (or interrupt). The debugger usually gives control to the user when the processor stops on a debug event, giving plenty of time to manually stop trace. So in many cases, it is not necessary to enable ptistop. However, it may be useful for debug events managed without user interaction, and when tracing is driven completely independently of debugging.
- The pto parameter simply reflects the current state of the processor trigger output (PTO) signal. It cannot be set.

11.5 Example Sequence of Commands

The following is an example sequence of commands to capture a trace around the point where the processor executes an instruction in the range 0x6000093c to 0x6000093f. This sequence can be packaged in a command file or executed interactively.

```
reset                          # reset TRAX
set pcstop0 0x6000093c:4       # trigger stop when PC in range 6000093c..6000093f
set syncper auto               # request more optimal synchronization setting
set postsize 10%               # capture 10% of TraceRAM past stop trigger
start                          # start tracing
status                         # show status
wait                           # wait until trigger hits and post-capture stops
save                           # save captured trace to file
status                         # show status again
```

Note that if the processor never executes anything in the selected PC address range, the wait command waits forever, or until the user presses Ctrl-C.

12. TRAX Trace Display Tool (xt-traxview)

The xt-traxview command-line tool decompresses processor execution traces captured and compressed using TRAX hardware. Such traces are usually first captured and saved under control of the xt-traxcmd tool described in Chapter 11. The resulting captured trace files are binary encoded and not readily human-readable (see Section 15.4). The xt-traxview tool converts these trace files into an annotated sequence of assembly instructions. The output is similar (though not identical) to that of the xt-trace command-line tool, documented in Chapter 18.

Note: TRAX 3.0 onwards, the trace file can be generated from xt-traxcmd, the TRAX software library, or from an external trace probe.

12.1 Synopsis

The xt-traxview tool accepts the following command-line options, each with both a short and a long name:

```
xt-traxview [-e|--executable filename[@offset]]*
            [-o|--output filename]
            [-s|--show options]
            [-a|--atid ATID]
             -t|--trace filename
```

xt-traxview displays a brief usage summary when you use the --help option.

12.2 Command-Line Arguments

This section discusses all command-line arguments of xt-traxview in detail.

12.2.1 Executable File (--executable filename)

This option specifies the target executable file that was running on the target when traces were captured. It is typically the file that the linker generates as the last step of the build process, and is in ELF format. This file provides the instruction opcodes, symbol table and other information necessary to display annotated assembly instructions.

The executable file specified here must correspond to what the processor was running when capturing the trace specified with the `--trace` option. Anything else will likely produce incorrect results: even mere recompilation of the same source code may result in a different executable. `xt-traxview` makes some partial effort to detect the use of an incorrect (mismatching) executable file. That is, it partially verifies whether branching recorded in the trace file correspond correctly to instructions in the executable file, and reports any discrepancies at the end of output (display of this trailer may be disabled with the `--show` option described in Section 12.2.4). Such discrepancies may be a sign that the wrong executable file was specified. However, discrepancies can also arise when code on the target is corrupted, or when the processor executes self-modifying code or code outside what is contained in the executable file.

If this option is not specified, normal disassembly is not output, and instruction counts are estimates only. Other items selectable by the `--show` option (see Section 12.2.4), including header and trailer information and disassembly lines not involving an opcode, are displayed.

If multiple independently linked images are present on the target, it may be necessary to specify this option multiple times, once for each image present (and possibly executed) at the time trace was captured. This can occur for example, with separate ROM and RAM images, with loadable libraries, or with operating systems that load separate application images and shared libraries such as in Linux.

Images should not overlap in memory; behavior in the presence of overlap is unspecified. For example, if you use a set of overlay images that load at the same address, specify only the overlay image that was present during trace capture. There is no provision to disassemble a trace stream that spans the execution of multiple images at the same address.

Libraries and images are normally expected to be loaded at the fixed address indicated in the ELF file. If a library load address is determined at load time, such as Linux shared libraries running on an Xtensa core, append the load address offset to the library filename given in the `-e` option, using the **`-e`** *`executable@offset`* syntax. Only the code offset is relevant here; if a library's code and data load addresses are independently determined, only provide the offset to the code's load address. For example, if a library `libc.so.0` is loaded at virtual address `0x20033000` [7], and the library's default load address is zero[8], the offset is thus `0x20033000-0=0x20033000`, so `xt-traxview` might be invoked with `-e libc.so.0@0x20033000` as one of its options.

7. In Linux running on an Xtensa core, library load addresses of process *PID* can be derived from looking at /proc/*PID*/maps while it is still executing.

8. The load address typically matches the lowest vaddr for a LOAD entry with non-zero filesz, in the output of `xt-objdump -p` *`executable.`*

12.2.2 Trace File (--trace filename)

This option specifies the trace file to display. This is normally a trace file captured and output by `xt-traxcmd` (see Chapter 11).

This option is required.

The file format of TRAX trace files is described in Section 15.4.

12.2.3 Output File (--output filename)

This option directs the annotated trace output to the specified file.

If this option is not specified, the annotated trace goes to standard output (`stdout`).

12.2.4 Display Options (--show options)

This option controls the general contents of the annotated trace output. The argument for this option consists of a sequence of alphabetic selectors. The selectors include:

`d`	normal disassembly
`s`	disassembly option: symbolic address lines on change of flow
`S`	disassembly option: symbolic operands
`h`	header (general information displayed at start of output)
`t`	trailer (statistics displayed at end of output)
`m`	decoded Nexus messages (see Section 15.2)
`M`	Nexus message option: trace offsets and message indices
`R`	Nexus message option: raw message bytes (hex and re-ordered binary)
`f`	machine readable condensed format showing changes in PC flow
`T`	display timestamps in disassembly/Nexus messages

The default output consists of normal disassembly, header, and trailer. In other words, not specifying this option is equivalent to specifying `--show dht` or `-sdht`.

For example, to output only disassembly with symbols (if available in the executable), specify `--show dsS` or `-sdsS`.

12.2.5 ATID (--atid ATID)

This option must be used with TRAX trace files produced by a multicore Xtensa system. It selects ATID of the core whose trace data is to be displayed.

This option may not be used with TRAX trace file produced by a single-core Xtensa system.

12.3 Example

The following example decodes the `mytrace.trax` trace file captured while the Xtensa processor was executing a program in the `myprog.out` ELF file and directs the output to a file named `decoded.txt`. Only disassembly and the statistics trailer are included in the output file.

```
xt-traxview -t mytrace.trax -e myprog.out -o decoded.txt -sdt
```

13. TRAX Control Library

The TRAX control library adds the ability to control and access TRAX hardware from a program running on an Xtensa processor. The library accesses the TRAX using the ERI interface (WER and RER instructions), allowing a core to trace its own execution. The library exports an Application Programming Interface (API) with functionality similar to what is provided by the `xt-traxcmd` software described in Chapter 11.

The following sections describe the software library API functions.

13.1 Control Library functions

The TRAX library API functions are segregated into the following four types, following the API provided by `xt-traxcmd` software:

- TRAX initialization
- Starting and stopping a TRAX session
- Setting and reading TRAX parameters
- Saving TRAX memory

All library functions use a TRAX context structure which contains information on the TRAX hardware and the current trace session. Details on the TRAX context are given in Section 13.1.1 and are not described with each API function.

13.1.1 TRAX initialization

This group contains trace initialization functions which set parameters of a trace session context. When starting the trace session for the first time, an initialization function needs to be invoked before any other library function. Thereafter, once the parameters of the context have been initialized, a call to *trax_reset* (discussed later) suffices to start a new trace session.

Note: Future versions of the library may provide other initialization functions to access TRAX over other specific interfaces (APB, for example).

int trax_context_init_eri (trax_context *context)

This function is used to initialize the TRAX context to access TRAX hardware over the ERI interface. It reads the various TRAX registers and sets up important information such as the size of the RAM in which the captured trace is stored (i.e. the TraceRAM), the TRAX PC version, and flags for debugging and other variables that are useful for returning the captured trace (compressed) back to the user, who can then use `xt-trax-view` to observe the uncompressed trace.

returns: 0 if successful, -1 if unsuccessful, -2 if the TraceRAM size is incorrect

13.1.2 Starting and Stopping a TRAX Session

This group contains functions that are used to start and stop a TRAX session, along with reset of TRAX parameters.

int trax_start (trax_context *context)

Starts tracing with the current parameter setting. If tracing is already in progress, an error is reported via return value. Otherwise, tracing starts and any unsaved contents of the TraceRAM are discarded.

returns: 0 if successful, 1 if trace is already active, -1 if unsuccessful.

int trax_stop_halt (trax_context *context, int flags)

This command initiates a stop trigger or halts a trace session based on the value of the flag parameter passed. If stop trigger is initiated, any selected post-stop-trigger capture proceeds normally. If the trace capture was not in progress, or if a stop trigger was already triggered, the return value indicates an error.

flag: Used to differentiate between stopping TRAX (based on trigger) or halting it by not capturing trace beyond trigger. A zero value stops the trace based on trigger, and a value of 1 halts it.

returns: 0 if successful, 1 if already stopped, -1 if unsuccessful.

int trax_reset (trax_context *context)

This command resets the TRAX parameters to their default value. Internally, it means resetting the different TRAX registers such as control, address, data, trigger, delay and match registers. This command also resets parameters of the context that deal with tracing mechanism so as to enable multiple tracing sessions by just performing a reset.

returns: 0 if successful, -1 if unsuccessful.

13.1.3 Setting and Reading TRAX Parameters

TRAX parameters allow the user to configure a TRAX session. This includes setting the synchronization message frequency, the TraceRAM boundaries within which the trace is intended to be captured, the stop point, external triggers, etc. Before these parameters are set the trace session context should be initialized as explained in Section 13.1.1

int trax_set_ram_boundaries (trax_context *context, unsigned startaddr, unsigned endaddr)

This function determines which part of the TraceRAM will be used during a trace session. The trace is collected in the TraceRAM portion determined by the start and the end address (word aligned). The minimum permissible difference between the start and the end addresses is 64 bytes. By default, if not initialized or set incorrectly, the start and end addresses cover the entire TraceRAM: *startaddr* is set to zero and *endaddr* is set to TraceRAM_words (the size of the TraceRAM in words) minus one.

startaddr: TraceRAM start address (in words) used for tracing.
Can be any value between 0 - (TraceRAM_words - 1).

endaddr: TraceRAM end address (in words) used for tracing.
Can be any value between 0 - (TraceRAM_words - 1).

returns: 0 if successful,
-1 if unsuccessful,
-2 if start and end address are specified incorrectly.

TraceRAM_words is the size of the TraceRAM in words (4 bytes per word).

Note: This function always returns success if the memory shared option is not configured, in which case the start and end addresses always cover the entire TraceRAM.

int trax_get_ram_boundaries (trax_context *context, unsigned *startaddr, unsigned *endaddr)

This function reports the start and end addresses of the TraceRAM (word aligned).

startaddr: the start address; default is zero.

endaddr: the end address; default is (TraceRAM_words - 1).

returns: 0 if successful, -1 if unsuccessful.

TraceRAM_words is the size of the TraceRAM in words (4 bytes per word).

Note: When the memory shared option is not configured, this function always reports *startaddr* and *endaddr* as their default values, covering the entire TraceRAM.

int trax_set_ctistop (trax_context *context, unsigned value)

This function selects the stop trigger via cross-trigger input.

value: 0 = off (reset value), 1 = on

returns: 0 if successful, -1 if unsuccessful

int trax_get_ctistop (trax_context *context)

This function reports whether the stop-trigger via cross-trigger input is off or on.

returns: 0 if off, 1 if on, -1 if unsuccessful

int trax_set_ptistop (trax_context *context, unsigned value)

This function selects the stop trigger via processor-trigger input.

value: 0 = off (reset value), 1 = on

returns: 0 if successful, -1 if unsuccessful

int trax_get_ptistop (trax_context *context)

This function reports whether the stop-trigger via processor-trigger input is off or on.

returns: 0 if off, 1 if on, -1 if unsuccessful

int trax_get_cto (trax_context *context)

This function reports cross trigger output state.

returns: 0 if `CTO` bit is reset, 1 if `CTO` bit is set

int trax_get_pto (trax_context *context)

This function reports processor trigger output state.

returns: 0 if `PTO` bit is reset, 1 if `PTO` bit is set

int trax_set_ctowhen (trax_context *context, int option)

This function selects the condition that asserts the cross trigger output.

option: 0 = off (reset value), 1 = ontrig, 2 = onhalt

returns: 0 if successful, -1 if unsuccessful

int trax_get_ctowhen (trax_context *context)

This functions reports the condition that asserted the cross trigger output. It can be any of: ontrig, onhalt, or off.

returns: 0 if off, 1 if ontrig, 2 if onhalt, -1 if unsuccessful

int trax_set_ptowhen (trax_context *context, int option)

This function selects the condition that asserts the processor trigger output.

option: 0 = off (reset value), 1 = ontrig, 2 = onhalt

returns: 0 if successful, -1 if unsuccessful

int trax_get_ptowhen (trax_context *context)

This functions reports the condition that asserted the processor trigger output. It can be any of: ontrig, onhalt, or off.

returns: 0 if off, 1 if ontrig, 2 if onhalt, -1 if unsuccessful

Note: To read more about types of trigger interfaces, refer to Section 11.4.6.

int trax_set_syncper (trax_context *context, int option)

This function selects the trace synchronization message period. If ATEN enabled, i.e. ATB output is enabled, synchronization message period cannot be set to off. Also, if no TraceRAM is present and ATEN is enabled, the sync period is set to its reset value i.e. 256.

option: 0 = off, 1 = on, -1 = auto, 8, 16, 32, 64, 128, 256 (reset value)

returns: 0 if successful, -1 if unsuccessful, -2 if incorrect arguments

Note: A value of 1 (“on”) ensures that the synchronization message frequency is internally set to be optimal. So, if the user is not sure what frequency of messages is optimal for the tracing session, it is recommended that the option be set to 1.

int trax_get_syncper (trax_context *context)

This function reports the synchronization message period.

returns: value of sync period, 0 if off, -1 if unsuccessful.

int trax_set_pcstop (trax_context *context, int index, unsigned long trig_start, unsigned long trig_end, int flags)

This function selects the stop trigger via PC match. Specifies the address or the address range to match against the program counter. Trace stops when the processor executes an instruction matching the specified address or range.

index: Indicates the number of stop trigger (currently there is only one, so must always be 0).

trig_start: Start range of the address at which the stop trigger should be activated.

trig_end: End range of the address at which the stop trigger should be activated.

flags: If non-zero, this inverts the range, i.e. trace stops when the processor executes an instruction that does not match the specified address or range.

returns: 0 if successful, -1 if unsuccessful, -2 if incorrect arguments (unaligned).

Note: For the current version of the TRAX control library, the *trig_end* can be set only 31 bytes away from the *trig_start* and the total range, i.e. *(trig_end* - *trig_start* + 1) should always be a power of two.

int trax_get_pcstop (trax_context *context, int *index, unsigned long *trig_start, unsigned *trig_end, int *flags)

This function returns the address or the address range which causes the trace to stop when the Program counter lies inside its values.

index: Contains index of the stop trigger (currently will always contain a 0).

trig_start: Points to start range of the stop trigger.

trig_end: Points to the end range of stop trigger.

flags: Contains information which indicates whether the pc stop range is inverted or not.

returns: 0 if successful, -1 if unsuccessful.

int trax_set_postsize (trax_context *context, int count, int unit)

This function is used to set the amount of trace to be captured past the stop trigger.

count: Contains the count of units (instructions or bytes) to be captured post trigger. If 0, it implies that this is off.

unit: Unit of measuring the count. 0 is bytes (in word size, i.e. multiples of 4), 1 is instructions, 2 is percentage of the TraceRAM.

returns: 0 if successful, -1 if unsuccessful, -2 if incorrect arguments.

Note: If the amount of trace to be captured post trigger is indicated in terms of bytes, i.e. *unit* is set to 0, *count* must always be greater than or equal to 32. This is because there is always some trace captured post the stop trigger point (which corresponds to the last synchronization message). Specifying less than 32 bytes would only display the last synchronization message and therefore the bytes read post trigger in the *trax_get_postsize* (as discussed below) would be zero

int trax_get_postsize (trax_context *context, int *count, int *unit)

This function reports the amount of TraceRAM in terms of the number of instructions or bytes captured post the stop trigger.

count: Will contain the count of units (instructions or bytes) post the stop trigger. If trax_set_postsize sets postsize in terms of bytes which are less than 32, then *count* would contain zero.

unit: Will contain information about the events that are counted. 0 implies that the TraceRAM words consumed are counted and 1 implies that the target processor instructions executed and exceptions/interrupts taken are counted.

returns: 0 if postsize was got successfully, -1 if unsuccessful.

13.1.4 TRAX Save Routines

The save routines are used to read back the compressed trace information from the TraceRAM (if it exists). Since embedded systems lack allocating large amounts of memory, the user must be able to read the trace in chunks of bytes. This requires the state of the trace amount read by the user inside the trace context.

When the trace is collected inside a TraceRAM or any other off-chip repository, ATB, etc., there must be a mechanism to return the compressed data to the user. If there is enough memory available on the embedded system, the user can just allocate a large size (using `malloc()`, for example), read the data from the memory and copy it (using `memcpy()`, for example).

In case the memory available is very limited, the user can prefer to read the v contents in chunks of bytes, the size of which is provided by the user. The first 256 bytes of data inside the RAM would be the header information which highlights information about the amount of trace captured, the time stamp etc. and is useful for initial debugging of the captured trace. The software must have some state information about the amount of trace already read by the user, the amount of trace to be read, etc. This state information is also maintained inside the context.

Note: The header of the trace session contains information regarding the session and therefore the first read from the TraceRAM must include all bytes that correspond to the header, i.e. the minimum chunk size should be at least the size of the header.

int trax_get_trace (trax_context *context, void *buf, int num_bytes)

This function returns a chunk of bytes of the trace data from the current pointer, which is maintained in the trace context.

Trace data is an array of bytes consisting of the header and the contents of the TraceRAM.

The first call would start from the beginning of the header and would progress the pointer based on the *num_bytes* passed. Since the header has to be read atomically, the first call should have its *num_bytes* at least as much as the size of the header.

buf:	Buffer that is allocated by the user. All the trace data read is placed in this buffer, which can then be used to generate a tracefile.
num_bytes:	Indicates the bytes the user wants to read. The first invocation needs this to be at least the size of the header i.e. 256.
returns:	Bytes actually read during the call to this function. 0 implies that all the bytes in the trace have been read, -1 if unsuccessful read/write of registers or memory, -2 if trace was active while this function was called, or -3 if user enters *num_bytes* less than the size of the header in the first invocation.

13.2 Library Example

The following code highlights the way the TRAX Software Library is intended to be used. This example initializes the trace session context, sets up several parameters such as TraceRAM start and end address range, the tracing stop point, the synchronization message frequency, the postsize i.e. the amount of tracing to be done once the stop trigger is reached and eventually saves the trace data in a tracefile.

Note: To link the program, the `-ltrax` linker or `xt-xcc` option is required to include the `libtrax.a` library.

```
#include <xtensa/trax.h>
```

/* Function that prints hello world in a loop */

```
int foo ()
{
     int i;

     for (i = 0; i < 100; i++) {
          printf ("Hello world: %d\n", i);
     }

     return 0;
}
```

/* The trax_save() function will save the contents of the TraceRAM into the file passed as an argument. This function is used assuming that the target is able to allocate memory that is enough to fit all the contents of the TraceRAM. Internally this function calls the trax_get_trace() function.*/

```
int trax_save (trax_context *context, char *file)
{

     int tfile, ret_val = 0;

     char buf[256];

     tfile = open (file, O_CREAT | O_RDWR | O_APPEND, S_IRUSR |
                    S_IWUSER | S_IRGRP | S_IROTH);

     if (tfile < 0)
          return -1;
```

/* While there are unread bytes in the TraceRAM, allocate a buffer and read the contents of the TraceRAM into this buffer. */

```
     while ((ret_val = trax_get_trace (context, buf, 256)) > 0)
     {
          if (write (tfile, buf, ret_val) != ret_val)
               return -1;
     }

     close (tfile);
```

```
        return 0;
    }


    int main ()
    {
        int trig_flag= 0, trig_index = 0;

        int syncval;

        int postsize_count = 0, postsize_unit = 0;
```

/* Specify the start address and the end address. In the current test case, we assume that the size of the TraceRAM is 4kB, i.e. 1024 words. So, the permissible values of start and end addresses would lie in the range of 0 - 1023, i.e. 0x0 - 0x3ff.*/

```
        unsigned startaddr = 0x14b;

        unsigned endaddr = 0x2dc;

        unsigned long pcstart = 0, pcend = 0;
```

/* Initializing the trax context is the first step */

```
        trax_context context;

        trax_context_init_eri (&context);

        trax_reset (&context);
```

/* The stop trigger is placed on the entry of the function "foo". The appropriate start and end ranges of the stop trigger are specified & verified. */

```
        trax_set_pcstop (&context, 0, (unsigned int)(&foo),
                         (unsigned int) (&foo) + 3, 0);

        trax_get_pcstop (&context, &trig_index, &pcstart, &pcend,
                         &trig_flag);

        printf ("Get pcstop: index: 0x%x, pcstart: 0x%x, pcend: 0x%x,
                 flags:0x%x\n", trig_index, (int)pcstart, (int)pcend,
                 trig_flag);
```

/* The start and end addresses are set. If incorrect values, they are set to their default values. This can be verified by the corresponding get functions. */

```
        int ret = trax_set_ram_boundaries (&context, startaddr, endaddr);

        if (ret < 0)
             printf ("Start or end addresses set incorrectly\n");

        trax_get_ram_boundaries(&context,&startaddr,&endaddr);

        printf("Startaddr:0x%x,endaddr:0x%x\n",startaddr,endaddr);
```

/* Synchronization period is set to 8, i.e. 1 synchronization message is emitted for every 8 messages emitted by the traceport. */

```
        trax_set_syncper (&context, 8);

        syncval=trax_get_syncper(&context);

        printf("Sync period set : %d\n", syncval);
```

/* The postsize indicates the amount of tracing to be done post the stop trigger. Here, it is set to 20% of the TraceRAM. */

```
        trax_set_postsize(&context,20,2);

        trax_get_postsize(&context,&postsize_count,&postsize_unit);

        printf ("Get postsize: count: %d, unit: %x\n", postsize_count,
                 postsize_unit);
```

/* After setting all the parameters and observing that all have been set appropriately, the tracing is started. */

```
        trax_start(&context);

        foo ();
```

/* The captured trace can be saved in the temp_tracefile, using the trax_save utility. */

```
        if (trax_save (&context, "temp_tracefile") < 0)
             return -1;

        return 0;
   }
```

14. TRAX Hardware

This chapter discusses the TRAX Module hardware.

14.1 Overview

Figure 1–1 shows an overall picture of the Debug module. Following are the main components relevant to TRAX:

- Traceport
- TraceCompressor
- TraceRAM
- ATB interface
- CrossTrigger and ProcessorTrigger interfaces

The TraceCompressor is the main component of TRAX hardware. Observing the Xtensa processor's Traceport, the TraceCompressor generates a stream of trace data compressed according to the Nexus standard (other than the items detailed in Section 15.3). The compressed trace is written into the TraceRAM through a standard SRAM interface. The TraceCompressor includes all of the logic that controls reading and writing the TraceRAM.

The TraceCompressor also houses various registers that are used to control and monitor the capture of trace. The registers are programmable through the chain of software and hardware components shown in Figure 1–1. The chain culminates at the Access Port interface to the TraceCompressor.

The TraceCompressor also contains trigger interfaces. Triggers are used to stop the capture of trace. Trigger inputs stop tracing in the TraceCompressor in question, whereas trigger outputs initiate events in other logic, including stopping tracing in TraceCompressors connected to other Xtensa processors. There are two types of trigger interfaces: the processor trigger interface (aka. the break interface), and the cross-trigger interface (see Section 14.6 for detailed information).

The components and interfaces of TRAX hardware are described in the following subsections.

14.1.1 Configurability

Following are the components of TRAX that are configurable:

- The size of the TraceRAM.
- Whether the TraceRAM is to be shared with other agents or not. If it is to be shared, the TraceRAM interface gets an additional signal `TraceMemReady`, which is used for arbitration.
- Whether trace is to be output on an ATB interface.
- Whether timestamps can be appended to the end of TRAX messages.

14.2 Traceport

The Traceport provides information reflecting the internal state of the processor during normal operation. Refer to the Traceport chapter of this guide for more details.

The `PDebugEnable` input (to the core) enables the Traceport logic, and the other signals (`PDebug***` outputs) provide the state of the core. Among other conditions, the Debug module turns on the Traceport whenever tracing is activated. Data trace is not supported by TRAX and hence the `PDebugLS***` buses are unused by the Debug module.

To allow for wiring delays from the core, the Xtensa flops the Traceport signals at output. This means that trace data creation happens in the cycle after the W-stage of the core pipeline.

14.3 TraceCompressor

As mentioned in Section 14.1, the TraceCompressor generates compressed trace according to IEEE-ISTO 5001™, as specified in *The Nexus 5001™ Forum Standard for a Global Embedded Processor Debug Interface*. The TraceCompressor does this by decoding the information on the Traceport every cycle, and maintaining information such as the number of instructions the processor has executed, the current PC, the last taken branch target, and so forth. It uses this information to assemble the message required for each traceable event.

The message then has to be packed according to the Nexus format of data bits (MDO) and auxiliary transmit-control bits (MSEO). TRAX uses six bits for MDO and two for MSEO, giving a byte-sized transmittal width. See Section 15.2.6 “Message Encoding” for a more detailed discussion. The total size of a message can vary from three to nine bytes (twenty bytes if timestamps are appended). Four bytes at a time are written to the TraceRAM. The TraceCompressor has an internal buffer to accommodate short-term bandwidth requirements.

14.3.1 Message Creation

Following are the Nexus messages pertinent to TRAX:

- Indirect Branch
- Indirect Branch With Sync
- Sync
- Correlation

An indirect branch is one where the target address is known only at runtime. Examples of conditional indirect branches are exception and interrupt; examples of unconditional indirect branches are JX, CALLXn, RET[W][.N], RFDE, RFE, RFI, RFWO, RFWU.

Synchronization messages are required by the spec at regular intervals. Resource full conditions are dealt with by issuing a sync message.

Trigger causes a correlation message. In the TRAX implementation, this is output with EVCODE = 0b001010.

14.3.2 Internal FIFO

A FIFO is used between the message creation logic and the trace sink path — i.e. TraceRAM or ATB. The sinks have bandwidth capacity of 32 bits per cycle. Message generation on the other hand could create up to 9 bytes (20 bytes if timestamps are appended) per cycle. The FIFO is used to deal with the peak bandwidth requirements of message creation.

With the addition of `TraceMemReady` or `ATREADY`, the sinks could have cycles where trace data is not accepted. So, the FIFO is also a queuing structure that mitigates the non-availability of sink bandwidth.

The FIFO has a size that is automatically derived from other configuration parameters. Regardless of how large it is, overflow is possible if Ready is deasserted sufficiently long. The TraceCompressor drops messages when a high water mark will be exceeded. It resumes message creation only when the internal FIFO drains to a low water mark.

The TraceCompressor restarts the trace stream with a synchronization message. The sync message has DCONT (the “discontinuity” bit) set, which allows xt-traxview to recognize that trace has been dropped.

Lost branch messages prevent the user from knowing the execution history of the processor. xt-traxview skips over these missing sections of trace with a message to the effect that PC cannot be reconstructed. The restart sync message has the full PC, so decompression can resume from that point.

In addition to branch messages, a correlation (i.e. trigger) point may be lost during an internal FIFO overflow. In these cases, the restart sync message has a special code that xt-traxview uses to tell the user that a trigger has been lost during the discontinuity.

14.4 TraceRAM

The TraceRAM is an on-chip repository of trace data of configurable size. It is a single-ported, single-cycle-latency memory. The interface from the Debug module to the TraceRAM is shown in the following table.

Table 14–56. TraceRAM Interface Signals

Name [Width]	Direction (with respect to TraceCompressor)	Description
`TraceMemAddr[log_mem_size-3:0]`	Output	TraceRAM address lines
`TraceMemData[31:0]`	Input	Input data lines from TraceRAM
`TraceMemEn`	Output	TraceRAM enable
`TraceMemWr`	Output	TraceRAM write line
`TraceMemWrData[31:0]`	Output	Data to be written to TraceRAM
`TraceMemReady`	Input	TraceRAM is not busy

`TraceMemReady` is present in the interface only if the TraceRAM is configured to be shared. This signal is used for arbitration, when other agents require access to the TraceRAM. The semantics of the signal are the same as that of ATREADY of the ATB interface viz. a read or write will only occur if both `TraceMemEn` and `TraceMemReady` are asserted in the same cycle. If `TraceMemReady` is deasserted, the TraceCompressor will retry the read or write in the next cycle.

The MemStartAddr and MemEndAddr registers may be modified to limit the portion of the TraceRAM used for trace data from *this* TRAX.

14.4.1 Implementation

With the exception of `TraceMemReady`, the interface from the TraceCompressor is a standard synchronous SRAM interface.

The *Xtensa Microprocessor Data Book* contains an appendix named “Connecting Memory to the Xtensa Processor”, which describes how to connect memories to the core. This appendix also includes waveforms that show the behavior of address, data, enable, write data and write enable. The TraceRAM is a single-ported, single-cycle-latency memory according to the definitions of the appendix. Use this appendix as a guide when designing the TraceRAM.

The implementation of the TraceRAM is determined by the developer. For example, if the size is small, it can be designed as a synthesizable register file. If the size is large, it may be more efficient to use a custom memory generated with a memory compiler.

The TraceRAM is not part of the TRAX deliverable. We include a behavioral model of TraceRAM in Verilog as part of the Reference Testbench, which allows RTL simulation of a system with TRAX. In this way, TraceRAM is similar to memories (such as the data RAM) connected to the Xtensa core.

14.4.2 Using the TraceRAM

Trace data is written to the TraceRAM when the TMEN bit of the TraxControl register is set. This includes writes resulting from normal tracing and reads/writes when reading the TraceRAM from JTAG, APB, or ERI. When this bit is cleared the TraceRAM becomes inaccessible. Note that in configurations with ATB absent, the TMEN bit is always set (for example, tied high).

14.4.3 Shared TraceRAM

As mentioned above, `TraceMemReady` is used for arbitration when the TraceRAM is configured to be shared. Deassertion of this signal prevents the TraceCompressor from writing internal FIFO data to the TraceRAM interface.

Variable TraceRAM portions are specified by each of the TRAX modules using the MemoryStart and MemoryEnd registers. To allow convenient muxing, the address output by each TraceCompressor is the full TraceRAM width, but only address values between MemoryStart and MemoryEnd will be output. As far as the writing and reading of trace data is concerned, the memory space remains a circular buffer between Start and End.

The customer designs the arbitration scheme and muxing hardware. Bandwidth allocations are controlled dynamically — i.e. by software writing the MemoryStart and MemoryEnd registers.

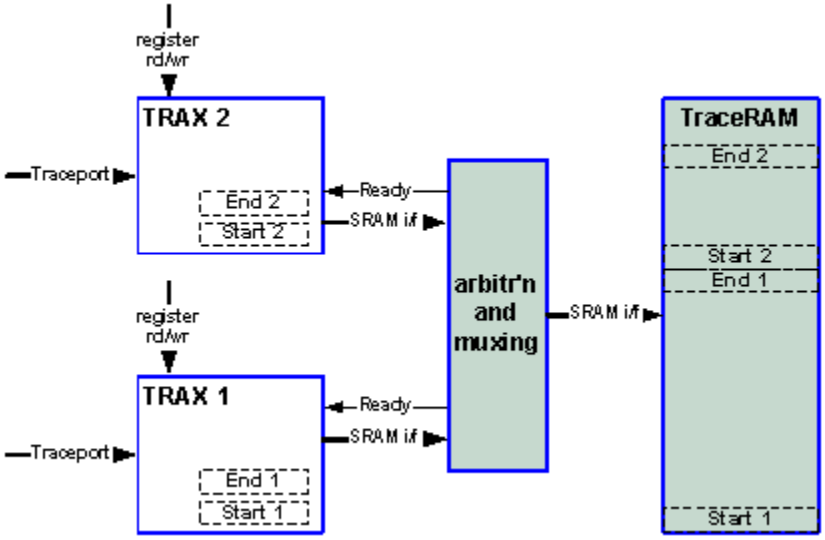


Figure 14–32. Sharing TraceRAM across Xtensa Cores

Users must be careful not to make MemStartAddr and MemEndAddr too close to each other. It may not be possible to get any useful disassembly if the size of the circular trace buffer becomes too small.

There is no requirement that MemStartAddr be less than MemEndAddr. If the start (MEMSTARTWORD) is indeed greater than the end (MEMENDWORD), the circular buffer containing the trace will simply straddle the TraceRAM address (TADDR) rollover boundary.

The MemStartAddr and MemEndAddr registers ignore writes to bits above log_mem_-size-3, so it is not possible to program an out-of-range TraceRAM address.

14.5 ATB Interface

TRAX has a master interface as specified by the AMBA 3 ATB protocol in the document *AMBA 3 ATB Protocol Specification v1.0 Issue A 2006/6/19*, publicly available from Arm.

14.5.1 ATB Versus TraceRAM

TRAX sends trace data to two sinks — the TraceRAM and ATB. The data is replicated — the TraceRAM does not act as an intermediate buffer to mitigate peak bandwidth requirements. The TraxControl register has two bits (ATEN and TMEN) that enable trace output to the ATB and TraceRAM respectively. When both enables are turned on *and* trace messages need to be dropped because of limited ATB bandwidth, these messages do not get recorded in the TraceRAM either — again because the data to the TraceRAM is a replica of that to ATB. To avoid this, the ATB enable can be turned off in a subsequent trace session, which would allow an unbroken trace stream to be recorded in the TraceRAM.

The trace enable control bits must not be changed in the middle of a trace session — i.e. while the TraceEnable bit (TREN) is set.

Note: The TMEN bit does not affect functionality for configurations that do not have ATB present; TMEN is tied high if ATB is absent.

14.5.2 ATB Interface Signals

The signals of the ATB interface are tabulated below.

Table 14–57. ATB Interface Signals

Name [Width]	Bits	Source	Comments
ATCLK	1	Input	Not implemented, clock within TraceCompressor used. ATCLK is an unconnected input pin.
ATCLKEN	1	Input	Used for TRAX clock gating i.e. enables TRAX (and Debug module) clock when asserted.
ATRESETn	1	Input	DebugReset within TraceCompressor used to reset ATB-related flops. ATRESETn is only used to quiet (i.e. data-gate) ATBYTES and ATDATA.
ATBYTES	2	Output	TRAX always writes words, so value is only “11” or “00”
ATDATA	32	Output	ATB data width is fixed at 32 bits (same as to TraceRAM)
ATID	7	Output	From TraxControl register bits [30:24] in TraceCompressor
ATREADY	1	Input	A transfer is successful only if both ATREADY and ATVALD are asserted in the same cycle
ATVALID	1	Output	ATB data valid
AFREADY	1	Output	Flush ready
AFVALID	1	Input	Flush valid

ATCLK and ATRESETn are signals provided for within the ATB specification. However, the TraceCompressor receives clock and reset separately — as signals that are part of the Debug module. ATCLK is unused within TRAX, and ATRESETn is used merely for data-gating of ATB outputs.

The Xtensa processor's ATB therefore is synchronized to the processor clock, CLK. For interface to a different clock domain, CoreSight IP — the Asynchronous ATB Bridge — is available. It is a component specifically meant to efficiently pass trace data across clock domains as described in the *CoreSight Components Technical Reference Manual Issue F 2008/4/18*.

ATCLKEN is the enable signal for the ATCLK clock domain. In Xtensa, it is used to clock-gate TRAX. When asserted, the clock to the Debug module (which includes TRAX) runs. However, when the TRAX clock is deasserted it might still run based upon (numerous) other conditions.

ATID comes from a programmable field of the TRAX Control register. Write to ATID while tracing is active (i.e. TRACT bit of TraxStatus register) will result in undefined behavior.

14.5.3 Flush

Up to 3 bytes of data could remain indefinitely in internal FIFO when the ATB is in normal use. Yet a downstream or off-chip trace collection buffer might want to issue a "trigger" point to secure all the trace data until that point. This represents an intermediate (and perhaps periodically issued) trigger, not a final trigger because the downstream trace collection logic might want to (for example) rebalance the priorities assigned to trace generating sources before continuing to gather more trace data. The flush functionality of the ATB addresses this need.

Flush is implemented according to the CoreSight architecture specification [*Version 1.0 ARM IHI 0029A 2004/9/29*]. As shown in Figure 14–33 (repeated from the ATB spec), the asserted AFVALID prompts the TraceCompressor to continually drain the internal FIFO. Upon the final transfer, AFREADY is asserted.

Because of its intermediate nature, flush needs to be recoverable. All data in the internal FIFO from the initial assertion of AFVALID is flushed. Subsequently generated trace data is retained in the FIFO, and will continue to be output after the flush is completed.

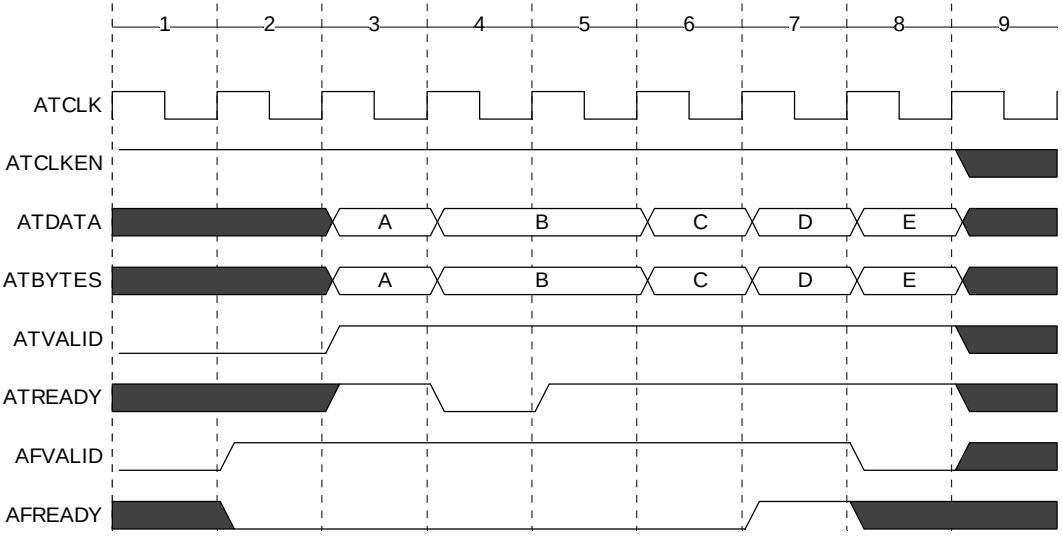


Figure 14–33. Flush Procedure *

(*copyright material repeated from AMBA ATB specification document)

14.5.4 ATB Unavailable

The following are situations during which ATB is unavailable. Under these circumstances, no trace data will be output. More importantly from a system perspective, TRAX will not perform a normal flush.

- Debug module is shut off (for Xtensa LX processors only; applies only to PSO configurations)
- Debug module is held in reset i.e. DReset is asserted
- ATEN is turned off
- Debug module is clock gated e.g. TREN is turned off

In the first three cases, ATVALID remains low for the whole period, and AFREADY remains high. In other words, if a flush is attempted by a trace sink, it will see an immediate response with no data. This is as allowed by the ATB spec.

The exception to this is clock gating. In this case, there will be no response from TRAX i.e. AFREADY low, so there is a danger of deadlock. To avoid this, tracing must be turned on.

When ATB is unused or unavailable, ATBYTES, ATID, and ATDATA will be undefined.

14.5.5 Direct Access to ATB and TraceRAM

Only the trace logic in TraceCompressor produces data that goes out on the ATB. In other words, it is not possible to do a write transfer through APB, ERI, or JTAG where the data ends up coming out of the ATB port.

The TraceCompressor does have the TraxData register that allows read/write to the TraceRAM from (for example) APB; however, reading and writing this register has no effect on ATB.

14.6 Triggers

As previously mentioned, triggers are used to stop the capture of trace. Trigger inputs stop tracing in the TraceCompressor in question, whereas trigger outputs initiate events in other logic, including stopping tracing in TraceCompressors connected to other Xtensa cores. There are two types of trigger interfaces: the processor trigger interface (also known as the TRAX break interface), and the cross-trigger interface.

14.6.1 Processor Trigger Interface

Table 14–58. Processor Trigger Interface

Name [Width]	Direction (WRT TraceCompressor)	Description
ProcessorTriggerIn	Input	Trigger in from core; is break-out with respect to core
ProcessorTriggerOut	Output	Trigger out to core; is break-in with respect to core

`ProcessorTriggerIn` indicates that Xtensa has taken a Debug interrupt, and can be used to initiate a stop trigger (PrimeTrigger). `ProcessorTriggerIn` is only recognized if it has been enabled (via PTIEN) and tracing has begun. If PrimeTrigger has already occurred for a different reason, then `ProcessorTriggerIn` assertion has no effect. `ProcessorTriggerIn` is edge-sensitive in that only at the cycle that Xtensa takes the Debug exception (and if enabled and if TRAX waiting for the PrimeTrigger) will it cause the PrimeTrigger.

`ProcessorTriggerOut` is an indication that the PrimeTrigger has happened, and it is used by OCD to cause Xtensa to take a Debug interrupt. `ProcessorTriggerOut` is asserted after PrimeTrigger has occurred if it is enabled by PTOWT, or after trace stop if enabled by PTOWS. `ProcessorTriggerOut` is deasserted upon clearing the TraceEnable bit (TREN) of the TraxControl register.

The PTOWT and PTOWS control bits are allowed to be turned ON before, during or well after the PrimeTrigger or trace stop. `ProcessorTriggerOut` will come ON even in the latter case. This is important to note because `ProcessorTriggerOut` is an edge-triggered input to OCD's interrupt logic, and an edge will be detected. In other words, the Break interrupt timing will not match the PrimeTrigger timing in this case.

Regardless of whether OCD takes the Break interrupt, a valid `ProcessorTriggerOut` is recorded into the DebugPendTrax bit of OCD's Debug Status register.

`ProcessorTriggerIn` and `ProcessorTriggerOut` are signals internal to the Debug module.

14.6.2 CrossTrigger Interface

Cross triggers are used to facilitate tracing multiple processors, or to coordinate the stopping of trace with non-processor related events. Figure 1–1 provides an example of how the cross trigger signals can be connected.

In contrast to the processor trigger interface, the cross trigger interface signals are all pins of Xtmem.

Table 14–59. Two Cross Trigger Interfaces

Name [Width]	Direction (WRT TraceCompressor)	Description
CrossTriggerIn	Input	Trigger in from other core or non-processor logic
CrossTriggerInAck	Output	Acknowledges CrossTriggerIn
CrossTriggerOut	Output	Trigger out to other core or non-processor logic
CrossTriggerOutAck	Input	Acknowledges CrossTriggerOut

`CrossTriggerIn` is level-sensitive, and causes the stop trigger (PrimeTrigger) only if it has been enabled – through CTIEN – and tracing has begun. If PrimeTrigger has already occurred for a different reason, then `CrossTriggerIn` assertion has no effect.

`CrossTriggerInAck` acknowledges that `CrossTriggerIn` was received.

`CrossTriggerOut` is a signal that indicates that the trace trigger has hit. `CrossTriggerOut` comes from the output of a flop, and goes high only if it is enabled — upon PrimeTrigger (CTOWT) or trace stop (CTOWS). It remains asserted high till the enable signals CTOWT and CTOWS are **both** turned off, or a new trace session is started — by software writing to the TraceEnable bit (TREN) of the TraxControl register.

Note that if the PrimeTrigger has already occurred, assertion of CTOWT or CTOWS does not cause `CrossTriggerOut` assertion, even if tracing is still ongoing — and indeed all the way until a new trace session is started.

Assertion of `CrossTriggerOutAck` by the CTI Module causes the TraceCompressor to deassert `CrossTriggerOut`. It is acceptable for external logic to keep `CrossTriggerOutAck` permanently tied high; this will not suppress `CrossTriggerOut` assertion. The only consequence would be that `CrossTriggerOut` assertion would be for one clock cycle only. Note that this is a different case than if `CrossTriggerOutAck` were to be asserted in the same cycle as `CrossTriggerOut`. In the latter case, `CrossTriggerOut` would be high for four cycles because of the synchronization flops in both directions.

14.6.3 Arm CTI Connection

`CrossTriggerIn/Out` is compatible with the CTI (cross trigger interface) as described in the *CoreSight Architecture Specification ARM IHI 0029A* publicly available from Arm.

Therefore, we provide the trigger/acknowledgment architecture shown in Figure 14–34 and tabulated below.

Table 14–60. CTI Signals

Name [Width]	Bits	Source	CTI Connection
`CrossTriggerOut`	1	TRAX	To ECTTRIGIN[4]
`CrossTriggerOutAck`	1	CTI	From ECTTRIGINACK[4]
`CrossTriggerIn`	1	CTI	From ECTTRIGOUT[4]
`CrossTriggerInAck`	1	TRAX	To ECTTRIGOUTACK[4]

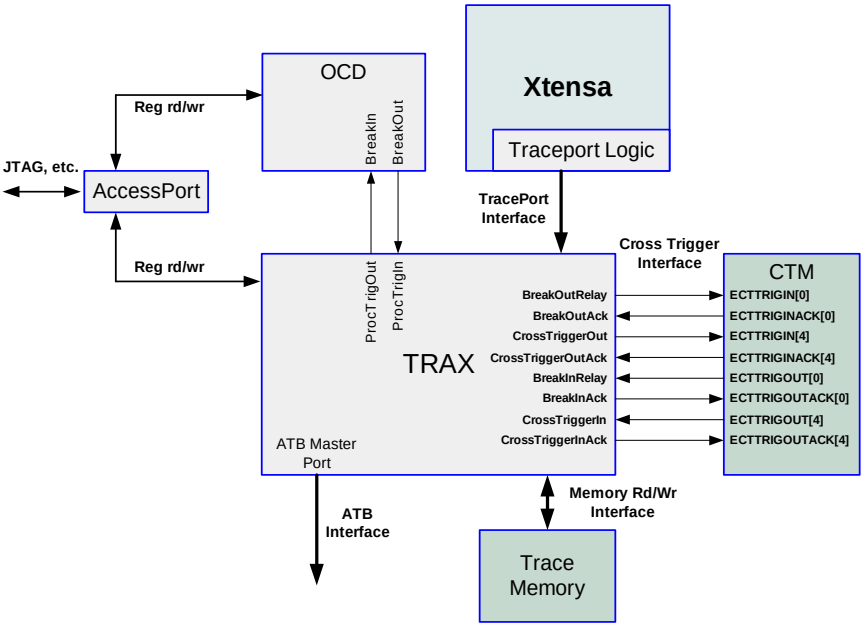


Figure 14–34. TRAX Connection to Arm CTM

The CTI requires that each trigger has a corresponding acknowledge signal, which is generated as shown in Figure 14–35. The purpose of this structure is to ensure that signals are able to be transmitted across widely-separated or asynchronous clock domains.

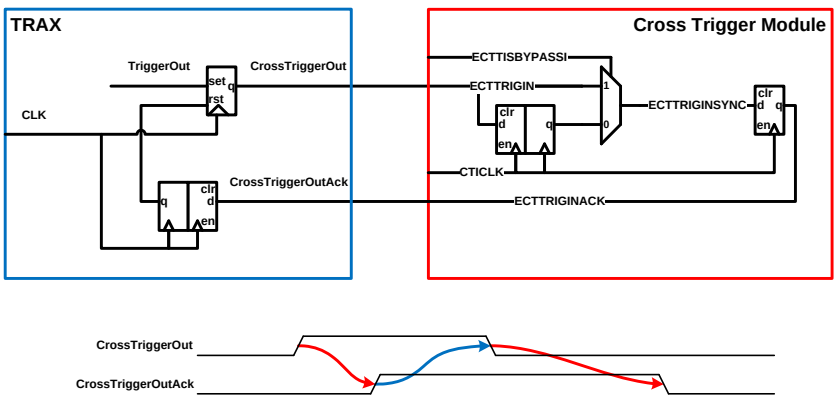


Figure 14–35. Cross Trigger Acknowledge Generation*

The signals ECTTIHSBYPASS, ECTTISBYPASSIN and ECTTISBYPASSACK are part of the CTI interface, but do not need to be driven from the trigger source/sink as per the Arm specification. Therefore Xtensa provides no equivalent outputs.

14.7 Timestamps

As mentioned in Section 14.1.1, TRAX hardware can be configured to append timestamps to TRAX messages. When configured, appending timestamps is software-enabled by setting the TSEN bit in the TRAX Control Register, i.e. TRAXCTRL[11]. Refer to Section 15.1.2 for TRAX Control Register details. The time value used for timestamping is the `DebugExtTime` input to Xtensa. This time value can be software-accessed by reading TRAX registers 17 and 16 in Table 15–62.

14.7.1 Timestamp Interface Signals

Table 14–61 shows the signal of the timestamp interface.

Table 14–61. Timestamp Interface

Signal Name	Width	Source	Comments
`DebugExtTime`	64	Input	Binary time value used by TRAX for timestamping. It is expected to be synchronous to the Xtensa clock.

Figure 14–36 shows an example of the global time network in a system where an Xtensa processor has its time input generated from CoreSight timestamp components (refer to the Arm CoreSight SoC-400 Technical Reference Manual). In this figure, the boxes in yellow are CoreSight timestamp components. The box in blue is an Xtensa instantiation showing the time connections inside its Debug module.

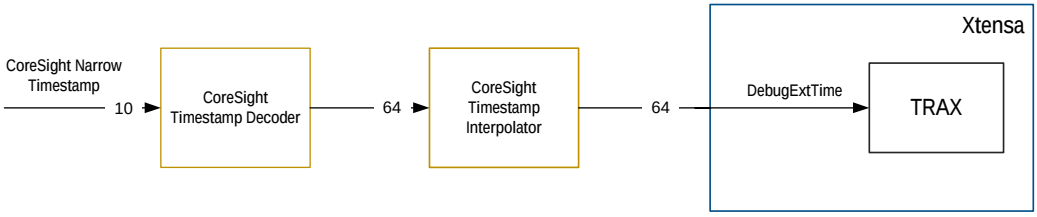


Figure 14–36. CoreSight-based Time Delivery

Note that you are not required to use CoreSight components. Any equivalent method that delivers a monotonically-increasing binary-coded 64-bit time value to the pins of an Xtensa processor is sufficient.

15. TRAX Programmer's Model

This chapter describes the functionality and operation of TRAX hardware from the perspective of someone writing trace management software to control it. Some of this information may also be useful to TRAX users needing more in-depth understanding.

15.1 TRAX Registers

TRAX registers are summarized in Table 15–62. Each register is 32-bits wide.

Table 15–62. TRAX Registers

Number	Name	Access	Reset Value	Description
0	TRAXID	RO	*(constant)*	Debug module ID Register
1	TRAXCTRL	R/W	0x00001080	Control Register
2	TRAXSTAT	RO	0x0000*mm*00	Status Register
3	TRAXDATA	R/W	*(TraceRAM entry)*	Data Register
4	TRAXADDR	R/W	0x00000000	Address Register
5	TRIGGERPC	R/W	0x00000000	Stop PC
6	PCMATCHCTRL	R/W	0x00000000	Stop PC Range
7	DELAYCOUNT	R/W	0x00000000	Post Stop Capture Size
8	MEMSTARTADDR	R/W	0x00000000	Start of trace in TraceRAM
9	MEMENDADDR	R/W	TraceRAM_words - 1	End of trace in TraceRAM
16	EXTTIMELO	RO	0x00000000	External time low portion - 32 LSBs of time value
17	EXTTIMEHI	RO	0x00000000	External time high portion - 32 MSBs of time value
10 .. 31	- -	--	--	*Reserved*

All registers except TRAXDATA can be read at any time. All writable registers may be written to while trace is not in progress. When trace is in progress, certain register writes must be avoided. See individual register descriptions for details.

Registers shown as reserved must not normally be accessed. However, they may be read at the end of a trace and stored in the trace file header (see Section 15.4).

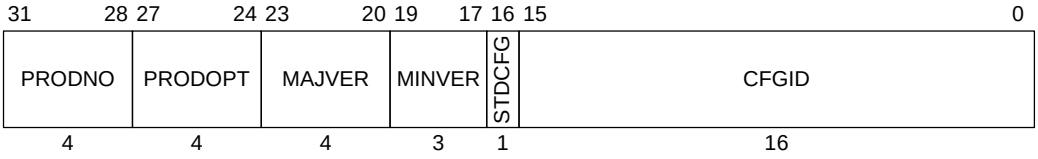
Unused bits (shown as '*' in Figure 15-37 through Figure 15-43) are reserved. These bits should be written either as zero or as they were read, and ignored when read.

Each register is described in more detail in the following subsections.

15.1.1 ID Register (TRAXID)

The TRAXID register is depicted in Figure 15-37. This register, which is the same as the OCDID register listed in Section 5.5, identifies the debug configuration.

Figure 15-37. ID Register Generic Layout



The ID register can be read at any time.

Table 15–63. ID Register Fields

Field	Width (bits)	Description
CFGID	16	Configuration ID. The value of this field is unique for each possible configuration of the Debug module.
STDCFG	1	"1" if it is a Cadence standard configuration. "0" if it is custom-configured.
MINVER	3	Minor version number. New minor versions are intended to be backward compatible. Starts at zero. For example, for version 2.0, MINVER is 0.
MAJVER	4	Major version number. New major versions are generally not backward compatible. Starts at one. Zero value is reserved. For example, for version 2.0, MAJVER is 2.
PRODOPT	4	Product specific options.
PRODNO	4	Product number (identifies device class/type). Product numbers currently assigned are: ■ 0 = TRAX ■ 1-15 = (reserved)

15.1.2 Control Register (TRAXCTRL)

The Control register, depicted in Figure 15-38 and detailed in Table 15–64, contains most of the TRAX configuration parameters and allows starting and stopping trace.

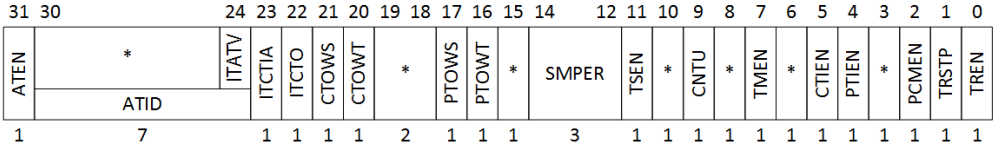


Figure 15-38. Control Register Layout

The Control register can be read at any time. It can be written when trace is not active (when TRAXSTAT.TRACT is zero). When trace is active, the only write allowed is to set the TRSTP bit or clear the TREN bit (while keeping all other fields unchanged) to trigger stopping the trace.

The value of this register never changes unless it is explicitly written to.

Table 15–64. Control Register Fields

Field	Reset Value	Description
ATEN	0	ATB enable. When set, trace output is sent out on the ATB interface (if ATB configured).
ATID / ITATV	7'h78	ATB source ID. Output directly as the ATID signal (if ATB configured). In integration mode, this field becomes ITATV (ATID[0]): the ATVALID output value. The ITMODE bit of the ITCTRL register controls this as described in Section 4.1.
ITCTIA	0	Integration mode: cross-trigger input acknowledge (CrossTriggerInAck).
ITCTO	0	Integration mode: cross-trigger output (CrossTriggerOut).
CTOWS	0	Cross-Trigger Output (CTO) is enabled when trace stop completes. This has no effect if CTOWT is set because CTO will already be high, See also CTOWHEN parameter in Section 11.4.6.
CTOWT	0	Cross-Trigger Output (CTO) is enabled when stop triggered. See also ctowhen parameter in Section 11.4.6.
PTOWS	0	Processor Trigger Output (PTO) is enabled when trace stop completes. This has no effect if PTOWT is set because PTO will already be high. See also ptowhen parameter in Section 11.4.6.
PTOWT	0	Processor Trigger Output (PTO) is enabled when stop triggered. See also ptowhen parameter in Section 11.4.6.

Field	Reset Value	Description
SMPER	0x001	Synchronization message period. This field selects whether and at what rate periodic synchronization messages are automatically emitted in the output trace. See `syncper` parameter in Section 11.4.5. When 0, no periodic synchronization messages are emitted. Otherwise (1 thru 6), synchronization messages are emitted every $2^{9\text{-SMPER}}$ messages. SMPER — Sync. Message Period 0 — none 1 — 1 every 256 2 — 1 every 128 3 — 1 every 64 4 — 1 every 32 5 — 1 every 16 6 — 1 every 8 7 — (*reserved*) See Selecting Periodic Synchronization Messages for more details.
TSEN	0	Enables embedding timestamp information in TRAX messages.
CNTU	0	Post-stop-trigger countdown units This bit selects what is counted by the countdown register DelayCount. The default is zero. See `postsize` parameter in Section 11.4.2. When 0, `DelayCount` counts down once for each 32-bit word written to the TraceRAM. When 1, `DelayCount` counts down once for each processor instruction executed and for each exception and interrupt taken.
TMEN	1	TraceRAM enable. Enables local trace memory. Always set (read as 0x1) in configurations without ATB.
CTIEN	0	Cross-trigger input enable. CTI causes Stop trigger (level-sensitive). See `ctistop` parameter in Section 11.4.6.
PTIEN	0	Processor trigger input enable. PTI causes Stop trigger (level-sensitive). See `ptistop` parameter in Section 11.4.6.

Field	Reset Value	Description
PCMEN	0	PC match enable. PC match causes Stop trigger. See `pcstop0` parameter in Section 11.4.1.
TRSTP	1	Trace stop. When this bit is set while `TRAXSTAT.TRACT` is set and `TRAXSTAT.TRIG` is clear, trace stop is triggered.
TREN	0	Trace enable. Setting this bit (from 0 to 1) while `TRAXSTAT.TRACT` is clear, initiates tracing: `TRAXSTAT.TRACT` is set, and writes to TraceRAM (and to ATB interface, if enabled) begin. Clearing this bit (from 1 to 0) once `TRAXSTAT.TRACT` is again clear, resets a portion of TRAX states, making it ready to setup a new trace: the following status register fields are all cleared (set to zero): TRIG, PCMTG, PTITG, CTITG, PTO, and CTO, and the Address register is set to the value of the `MEMSTARTADDR` register. Results of changing this bit while `TRAXSTAT.TRACT` is set are undefined. To forcefully stop trace immediately while `TRAXSTAT.TRACT` is set, it is best to follow this sequence: set TRSTP (not necessary if already set), clear `DELAYCOUNT` (not necessary if already zero), at which point `TRAXSTAT.TRACT` should be clear, so finally `TRAXCTRL.TREN` can be cleared. See Figure 15-39 and Section 15.5 for more details.

TRAX Tracing States

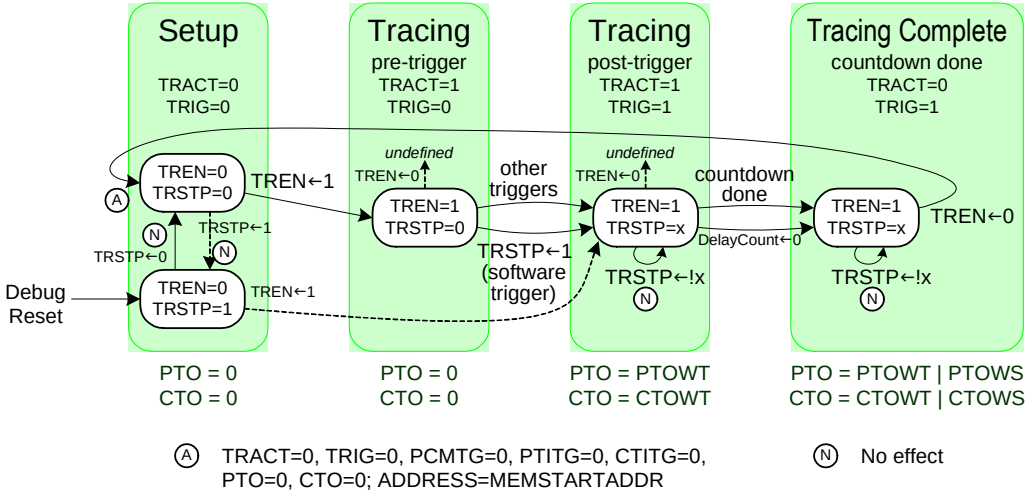


Figure 15-39. TRAX State Diagram

Dashed-line transitions are not necessary; it is recommended that they be avoided.

15.1.2.1 Selecting Periodic Synchronization Messages

The Nexus standard requires at least one synchronization message for every 256 messages emitted over a standard AUX connector. However, if trace only goes to the local TraceRAM (not to the ATB bus), TRAX does not require this for proper operation, so periodic synchronization messages are typically left disabled. This is because TRAX outputs trace in such a way that it is always possible to reconstruct the PC flow from the end of a trace to its beginning: traces always end with a synchronization message (at least in the case of local TraceRAM after trace is no longer active), not a branch with sync message; and all other messages provide differential types of PC addresses.

When enabling periodic synchronization messages, it is generally desirable to select a period that minimizes total synchronization overhead. This overhead consists of two parts: the synchronization messages themselves, and whatever initial portion of the captured trace (until the first synchronization message) cannot be processed (upon wrap-around of the circular TraceRAM)[9]. It is easier to compute the period by first dealing in bytes. Assume *p* is the (average) synchronization message period in TraceRAM bytes, *s* is the average synchronization message size in bytes, and *m* is the size of the TraceRAM in bytes. The total synchronization overhead *T* can be calculated as:

$$T = \frac{ms}{p} + \frac{p}{2}$$

Here we assume that the portion of captured trace that precedes the first synchronization message can never be processed. On average, that portion will be half of the synchronization period, hence the term *p*/2.

We want to know *p* given *s* and *m*, while minimizing *T*. The above is a simple quadratic equation that reaches its minimum when its derivative in *p* is zero:

$$-\frac{ms}{p^2} + \frac{1}{2} = 0$$

Solving for *p* gives the formula:

$$p = \sqrt{2ms}$$

Given *p* which is the period in bytes, we need to calculate *n*, the period in messages (including the synchronization message itself). If the average non-synchronization message length is *w*, we have:

$$n = \frac{p - s}{w} + 1$$

9. In TRAX, only indirect branch with synchronization messages prevent processing backwards (Direct Branch messages are not used), so the latter unprocessable trace fragment loss only occurs when periodic synchronization messages are enabled.

TRAX only supports values of *n* from 8 to 256 in powers of 2, so we have to choose the adjacent supported value that provides the least overhead. Solving for TraceRAM sizes *m*, *s* = 8, and *w* = 3, we get the best match *n* listed in Table 15–65. Only supported TraceRAM sizes are shown (see the MEMSZ field in Section 15.1.3 on page 255).

Table 15–65. Suggested Synchronization Message Periods

Syncs Needed	TraceRAM Size (bytes)	Optimal *n* (messages)	Best Match *n* (messages)	Overhead *T* (bytes)	TraceRAM Utilization
No	any	None	None	0	100%
Yes	128	1 in 11	1 in 16	46	64%
Yes	256	1 in 18	1 in 16	65	75%
Yes	512	1 in 27	1 in 32	91	82%
Yes	1 K	1 in 39	1 in 32	132	87%
Yes	2 K	1 in 57	1 in 64	182	91%
Yes	4 K	1 in 82	1 in 64	265	94%
Yes	8 K	1 in 117	1 in 128	363	96%
Yes	16 K	1 in 167	1 in 128	531	97%
Yes	32 K	1 in 238	1 in 256	726	98%
Yes	64 K and above	1 in 338+	1 in 256	1065+	98+%

15.1.3 Status Register (TRAXSTAT)

The status register, depicted in Figure 15-40 and detailed in Table 15–66, provides various status information that detail the progress of trace capture.

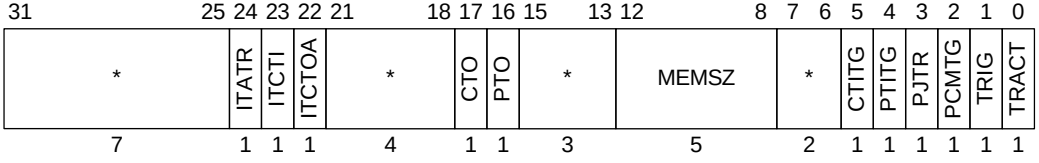


Figure 15-40. Status Register Layout

The status register can be read at any time, with no side-effect.

Table 15–66. Status Register Fields

Field	Reset Value	Description
TRACT	0	Trace active flag. This bit indicates whether trace is active, during which messages are output to TraceRAM. When 0, trace is inactive. It is safe to access the TraceRAM. When 1, trace is active. This bit is set automatically when trace is initiated i.e. Control.TREN transitions from 0 to 1, and cleared automatically when trace stops (see Figure 15-39 and Section 15.5).
TRIG	0	Trace stop trigger. This bit is cleared automatically when TRAXCTRL.TREN transitions from 1 to 0, and set once the stop trigger occurs. Possible stop trigger sources include PC match, cross-trigger input, processor trigger input, and manual trigger using the TRAXCTRL.TRSTP bit.
PCMTG	0	Stop trigger caused by PC match. This bit is set if the stop trigger was caused by a PC match. This bit is cleared automatically when TRAXCTRL.TREN transitions from 1 to 0.
PJTR	0	JTAG transaction result. Indicates success (0) or failure (1) of the preceding JTAG transaction.
PTITG	0	Stop trigger caused by Processor Trigger Input (PTI). This bit is set if the stop trigger was caused by PTI. This bit is cleared automatically when TRAXCTRL.TREN transitions from 1 to 0.
CTITG	0	Stop trigger caused by Cross-Trigger Input (CTI). This bit is set if the stop trigger was caused by CTI. This bit is cleared automatically when TRAXCTRL.TREN transitions from 1 to 0.
MEMSZ	mmmmm	TraceRAM size. The size of the TraceRAM as seen by the TRAX compressor is 2^{MEMSZ} bytes. TRAX only supports TraceRAM sizes from 512 bytes to 256 KB, which correspond to MEMSZ field values 9 thru 18. Although measured in bytes, the TraceRAM is only accessible as 32-bit words. The value reported here corresponds to what was configured in the Xtensa Processor Generator.

Field	Reset Value	Description
PTO	0	Processor Trigger Output This bit reflects the current value of PTO. It is cleared when TRAXCTRL.TREN transitions from 1 to 0, or when PTOWT or PTOWS bits that make it set are cleared. It is set while TRAXCTRL.TREN is set when either: TRAXCTRL.PTOWT is set and the stop trigger fires, or TRAXCTRL.PTOWS is set and trace output stops. PTO is an internal signal that is latched into OCD register bit DSR.DebugPendTrax when a TRAX trigger causes a debug interrupt.
CTO	0	Cross-Trigger Output This bit reflects the current value of CTO. It is cleared when TRAXCTRL.TREN transitions from 1 to 0, or when CTOWT or CTOWS bits that make it set are cleared. It is set while TRAXCTRL.TREN is set when either: TRAXCTRL.CTOWT is set and the stop trigger fires, or TRAXCTRL.CTOWS is set and trace output stops. CTO is then also cleared when the CrossTriggerOutAck signal is received.
ITCTOA	0	CrossTriggerOutAck input observation value used in integration mode.
ITCTI	0	CTI input observation value used in integration mode.
ITATR	0	ATREADY input observation value used in integration mode.

15.1.4 Data Register (TRAXDATA)

The Data register reflects the 32-bit TraceRAM word indexed by the Address register's TADDR field. Reading the Data register reads the selected TraceRAM word, and writing the Data register writes the selected TraceRAM word. Every access to the Data register from JTAG or APB automatically increments the Address register by one, so that it points to the next TraceRAM word. This is a post-increment: TRAXADDR is incremented *after* the access (see Section 15.1.5).

The Data register can only be accessed, whether for reads or writes, when trace is not active (when TRAXSTAT.TRACT is zero).

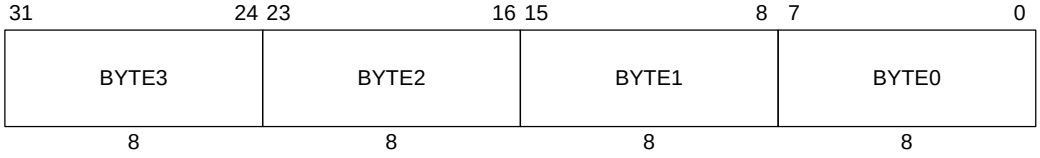


Figure 15-41. Data Register Layout

The TRAX compressor outputs to the TraceRAM a trace encoded in Nexus format as a stream of bytes. Four such bytes show up at a time in the data register, ordered from the earliest byte is in the least significant 8 bits of the data register to the most recent byte in the most significant 8 bits of the data register, as shown in Figure 15-41. Although this is nominally a little-endian ordering, it is independent of the target processor or host byte ordering.

Accesses to TRAXDATA might be ignored if the previous access to TRAXDATA is still in progress. Therefore, you must check the access response available on different access mechanisms, which are E and B bits for JTAG and the ERISTAT register. Also, to ensure safe access to TRAXDATA over the JTAG, poll the TRAXSTAT.PJTR bit first. In addition, TRAX address register auto-increments when reading TRAXDATA from JTAG or APB. Thus, you can check if the address register is incremented properly after each access.

The address register is not auto-incremented on any access to TRAXDATA (read or write) from the ERI. This is in part to deal with the fact that the RER instruction is speculative.

Reads to TRAXDATA return unpredictable results when the TMEN bit in TRAXCTRL is 0.

15.1.5 Address Register (TRAXADDR)

The address register, depicted in Figure 15-42 and detailed in Table 15–67, selects the TraceRAM word accessed by the data register and provides wrap-around statistics.

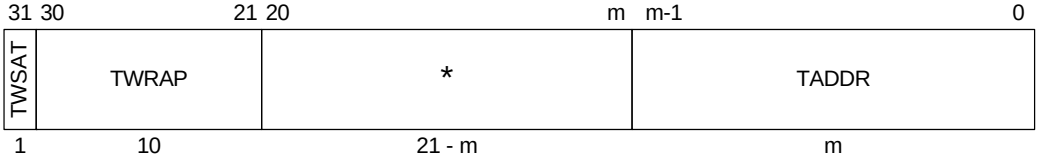


Figure 15-42. Address Register Layout

The address register points to a word in the TraceRAM. It is automatically post-incremented by JTAG or APB (but not ERI) accesses to the TRAXDATA register, and whenever a word is written to TraceRAM while actively tracing. These auto-increments update all fields of this register: wraparound of TADDR increments TWRAP, and overflow/wraparound of TWRAP sets the TWSAT bit; see Table 15–67.

The address register can be read at any time. It can only be written when trace is not active (when Status.TRACT is zero). Reading this register while trace is active reports the next TraceRAM location to be written by TRAX.

When trace stops, the TADDR field points to the first TraceRAM word past the captured trace. If TWRAP and TWSAT fields are all zero, the captured trace consists of all TraceRAM words from index MEMADDRSTART (or index zero, if absent) up to index TADDR-1, inclusive. Otherwise the captured trace consists of TraceRAM words from index TADDR to index TADDR inclusive (or to the end of memory, if absent), followed by words from index MEMADDRSTART (or index zero, if absent) up to index TADDR-1, inclusive.

Table 15–67. Address Register Fields

Field	Width (bits)	Description
TADDR	m	TraceRAM word address pointer. This field indexes 32-bit words (not bytes) in the TraceRAM. Thus, m = MEMSZ - 2. Cleared when TRAXCTRL.TREN goes from 0 to 1.
TWRAP	10	Wrap-around count. This field counts how many times TADDR wrapped around from the end to the beginning of the entire TraceRAM. It thus increments by the carry from TADDR increments, such that TWRAP and TADDR effectively work as one large counter. Together, TWRAP and TADDR report the total number of trace words written. Cleared when TRAXCTRL.TREN goes from 0 to 1.
TWSAT	1	Wrap-around overflow. This is essentially a saturating carry off the top of TWRAP. It is set when TWRAP and TADDR increment from an all ones value to an all zeroes value. It is not cleared by auto-increments of TADDR and TWRAP. Cleared when TRAXCTRL.TREN goes from 0 to 1.

15.1.6 Stop PC (TRIGGERPC)

The stop PC register holds the 32-bit address compared with the processor program counter when the PCMEN bit of the Control register is set. A match for a given executed instruction triggers trace stop (asserts the stop trigger).

Some number of least significant bits of this register may be ignored, according to the PCML field of the PCMATCHCTRL register. And the comparison is inverted if the PCMS field of PCMATCHCTRL.

This register can be read and written at any time. The exact timing of writes made while tracing is active (while TRAXSTAT.TRACT is set) is not defined.

This register is used to implement the pcstop0 parameter described in Section 11.4.1.

15.1.7 Stop PC Range (PCMATCHCTRL)

The PC match control register, depicted in Figure 15-43 and detailed in Table 15–68, controls the PC watchpoint trigger.

This register is used to implement the `pcstop0` parameter described in Section 11.4.1.

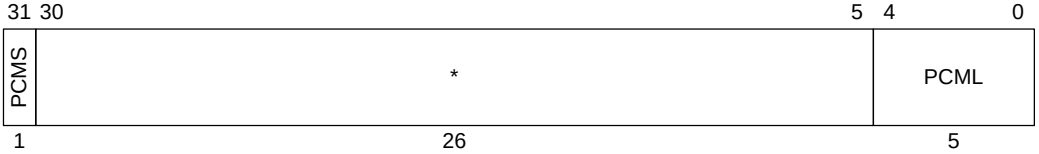


Figure 15-43. PC Match Control Register Layout

This register can be read and written at any time. The exact timing of writes made while tracing is active (while `Status.TRACT` is set) is not defined.

Table 15–68. PC Match Control Register Fields

Field	Width (bits)	Description [Long Name]
PCML	5	PC Match Mask Length. Indicates how many of the lower bits of the Trigger PC register to ignore.
PCMS	1	PC Match Sense. When 0, match when the processor's PC is in-range of the Trigger PC register and mask. When 1, match when the processor's PC is out-of-range of Trigger PC register and mask.

Thus:

```
PC_matched = Sense ^ ((TriggerPC | mask) == (TraceportPC | mask))
```

where:

```
mask = (1 << PCML) - 1
```

15.1.8 Post Stop Trigger Capture Size (DELAYCOUNT)

The `DELAYCOUNT` register counts events from the occurrence of the stop trigger until trace stops completely. The delay count register is decremented for every *event* occurrence (see below). Once the delay count reaches zero, the final stop phase is initiated. If this register is zero when trace starts, the stop trigger immediately initiates the final stop phase. In the final stop phase, a final synchronization message is emitted and all internally buffered messages are flushed to the TraceRAM.

Only the lower 24 bits of the register are significant. The upper 8 bits are reserved.

Implementation of the postsize parameter (see Section 11.4.2) uses this register.

This register can be read at any time. It can only be written when trace is not active (TRAXSTAT.TRACT is zero), or when writing a zero to it while trace is active for the purpose of causing trace to stop immediately.

The CNTU field of TRAXCTRL selects what *events* are counted. These events, or *units*, are either:

- TraceRAM words consumed (CNTU = 0), or
- target processor instructions executed and exceptions/interrupts taken (CNTU = 1).

The DELAYCOUNT register controls how much trace, if any, is captured following the trigger. Because the TraceRAM is circular, this register can be used to select whether to capture a TraceRAM worth of trace data:

a. before the trigger,
b. around the trigger,
c. after the trigger, or
d. some distance after the trigger (in which case the trigger's correlation message is lost because the trigger is no longer contained in the captured trace).

For a trigger at the end of the trace (case a), initialize DELAYCOUNT to zero. Placing the trigger's correlation message at other specific locations within the captured trace is most easily accomplished by counting down TraceRAM words. For a trigger at the start of the captured trace (case c), initialize DELAYCOUNT to the size of TraceRAM minus the maximum amount of trace output in the final stop phase (does not exceed 32 bytes). For a trigger at other positions within the captured trace (case b), initialize DELAYCOUNT to a proportional fraction of the value computed for trigger at start.

Note: The DELAYCOUNT register itself decrements during the delay phase, so it must be re-initialized appropriately at the beginning of every trace capture session.

15.1.9 Trace Memory Start Address and End Address

The MEMSTARTADDR and MEMENDADDR registers delimit the circular trace buffer area within TraceRAM. Each consists of only one field occupying the least significant *m* bits, i.e. the same width and position as TRAXADDR.TADDR. The upper bits (31:*m*) are read as zero and ignored on writes.

These registers are only writable if the TRAX "shared memory" option is configured.

At reset, MEMSTARTADDR is zero, and MEMENDADDR is $2^{MEMSZ-2}$-1, i.e one less than the number of 32-bit words in the TraceRAM. If the registers are not writable, they always return these values.

On initiating trace, TRAXADDR.TADDR is set to MEMSTARTADDR.

TRAXADDR.TADDR post-increments (upon DATA register accesses, and trace writes to TraceRAM) as follows:

```
if TADDR = MemEndAddr; wrap-around?
     TADDR ← MemStartAddr
     TWRAP ← TWRAP + 1
     if TWRAP = 0
          TWSAT ← 1
     fi
else
     TADDR ← TADDR + 1
fi
```

There is no requirement that MEMSTARTADDR be less than MEMENDADDR. If MEMSTARTADDR is greater than MEMENDADDR, this means that the circular buffer area containing the trace straddles the end/start of TraceRAM.

The effective size of the circular buffer area — that is, the span of bytes from MEMSTARTADDR to MEMENDADDR (inclusive) — must be at minimum 64 bytes. There is no alignment requirement (beyond the unavoidable 4-byte alignment).

15.1.10 Time Value Registers

Software may read TRAX register pair 17 and 16 (shown in Table 15–62) to access the time value used for timestamping. The ExtTime value (EXTTIMEHI concatenated with EXTTIMELO), which is read-only, reflects the synchronized value of the DebugExtTime input. It can be read using ERI, APB, or JTAG.

15.2 Trace Messages

The TRAX compressor hardware module generates compressed processor execution traces using data from the processor trace port. This section briefly describes the format of the resulting compressed trace data.

The output of the TRAX compressor follows the *IEEE-ISTO 5001™-2003* (Nexus) standard specification from the Nexus 5001™ Forum (www.nexus5001.org), with minor variations. The remainder of this chapter assumes familiarity with this standard.

Technically, the Nexus standard does not specify a file format. Instead, it specifies the *AUX port* (auxiliary port) as a set of signals for off-chip real-time trace, and an *AUX port protocol* for sending trace messages over this port. However, TRAX outputs trace messages into a local buffer rather than to an off-chip probe. What it stores in successive bytes of the buffer is equivalent to what would be output in successive cycles over an AUX port, except that any idle state AUX port values are omitted.

TRAX implements Program Trace using Traditional Branch Trace Messaging (BTM). Specifically, it implements the following Nexus Public Messages:

- Program Trace — Indirect Branch Message
- Program Trace — Synchronization Message
- Program Trace — Indirect Branch with Synchronization Message
- Program Trace — Correlation Message

Specific details on each of these message types are provided in the following sections.

The description of each message describes its structure, which consists of a sequence of packets, each packet being a sequence of bits divided into bit fields. The detailed encoding of this message structure is described next, in Section 15.2.6.

15.2.1 Indirect Branch Message

The indirect branch message, described in Table 15–69 for Xtensa NX processors and Table 15–70 for Xtensa LX processors, indicates a change in program flow.

A direct branch is one where the target address is encoded into the instru ction stream. An indirect branch is one that computes the target address using a register or other indirection, and cannot readily be obtained by simply decoding the instruction. TRAX emits indirect branch messages for both direct and indirect branches.

Specifically, when `Status.TRACT` is set, TRAX emits an indirect branch message:

For Xtensa NX processors:

- with `B-TYPE=0` for any instruction that results in a change in program flow (that doesn't fall-through to the next instruction)
- with `B-TYPE=2` for any zero-overhead loopback from `LEND` to `LBEG`
- with `B-TYPE=1` for any non-OCD taken exception or interrupt; in this case, `U-ADDR` reports the exception or interrupt vector address
- with `B-TYPE=6` for any OCD taken exception; in this case, `U-ADDR` reports the exception or interrupt vector address
- with `B-TYPE=4` for any OCD exception return.

For Xtensa LX processors:

- with B-TYPE=0 for any instruction that results in a change in program flow (that doesn't fall-through to the next instruction)
- with B-TYPE=0 for any zero-overhead loopback from LEND to LBEG
- with B-TYPE=1 for any taken exception or interrupt; in this case, U-ADDR reports the exception or interrupt vector address

However, if a synchronization message is due at the same time as the indirect branch message, an indirect branch with synchronization message will be emitted instead. This occurs when the indirect branch coincides with the start of a trace capture session, or with a periodic synchronization message when Control.SMPER is non-zero, or when I-CNT is about to overflow its 10-bit range. It does not occur for the last message sent, which is always a synchronization message, when trace stops gracefully.

Notes:

- Zero-overhead loopbacks are treated as if a branch operation is part of the last instruction in the loop. In that case, the I-CNT field of the indirect branch message does not include the last instruction of the loop. In particular, the I-CNT field is zero when reporting zero-overhead loopback of a loop containing a single instruction.

Table 15–69. Indirect Branch Message Format for Xtensa NX Processors

Packet	Field	Size (bits)	Value	Description
1	TCODE	1	0	Message type code
	B-TYPE	3		Branch type: 0 = normal branching, 1 = non-OCD exception or interrupt, 2 = Loopback, 4 = OCD exception Return, 6 = OCD exception
	I-CNT	1-10		Number of instruction bytes executed since the last branch or sync. message (correlation messages do not affect I-CNT). Does not include the instruction causing this branch (B-TYPE=0) or that took the exception/interrupt and thus did not commit (B-TYPE=1).
2	U-ADDR	1-32		Unique portion of the branch target address. Computed as the exclusive-or (XOR) of the branch target address and of the last target address or program counter indicated in a message (all messages except correlation messages provide either a target address or a current program counter).
3	TSTAMP	0-64		Time value

Table 15–70. Indirect Branch Message Format for Xtensa LX Processors

Packet	Field	Size (bits)	Value	Description
1	TCODE	6	4	Message type code
	B-TYPE	1		Branch type: 0 = normal branching, 1 = exception or interrupt
	I-CNT	1-10		Number of instruction bytes executed since the last branch or sync. message (correlation messages do not affect I-CNT). Does not include the instruction causing this branch (B-TYPE=0) or that took the exception/interrupt and thus did not commit (B-TYPE=1).
2	U-ADDR	1-32		Unique portion of the branch target address. Computed as the exclusive-or (XOR) of the branch target address and of the last target address or program counter indicated in a message (all messages except correlation messages provide either a target address or a current program counter).
3	TSTAMP	0-64		Time value

For example, given the following instruction sequence executed from address 0x100:

```
0x100:  j     0x120
 :
0x120:  add   a2, a3, a4
0x123:  s16i  a2, a5, 0
0x126:  call0 0x150
 :
0x150:  ...
```

The J and CALL0 instructions normally result in indirect branch messages. The message emitted for the CALL0 instruction above is constructed with:

- B-TYPE = 0 (not an exception or interrupt)
- I-CNT = 6 (total size of add and s16i instructions)
- U-ADDR = 0x70 (0x120 xor 0x150)
- TSTAMP = DebugExtTime (Optional current time value)

15.2.2 Indirect Branch with Synchronization Message

The indirect branch with synchronization message, described in Table 15–71 (for Xtensa NX processors) and Table 15–72 (for Xtensa LX processors), indicates a change in program flow. It is very similar to the indirect branch message, but provides a full target address instead of a relative one, and adds the synchronization message's DCONT field to signal program flow discontinuities.

TRAX emits this message when both an indirect branch and a synchronization message are due to be emitted in the same cycle. The conditions under which this happens are already described in Section 15.2.1.

The DCONT field is described in Section 15.2.3.

Table 15–71. Indirect Branch with Synchronization Message Format for Xtensa NX

Packet	Field	Size (bits)	Value	Description
1	TCODE	2	3	Message type code
	DCONT	1		Discontinuity: 0 = follows another message, 1 = new sequence
	B-TYPE	3		B2 = Loopback, 4 = OCD exception Return, 6 = OCD exception branch type: 0 = normal branching, 1 = non-OCD exception or interrupt,
	I-CNT	1-10		Number of instruction bytes executed since the last branch or sync. message (correlation messages do not affect I-CNT). Does not include the instruction (pointed to by F-ADDR) causing this branch (B-TYPE=0) or that took the exception/interrupt and thus did not commit (B-TYPE=1). If DCONT is 1, this field is undefined by Nexus. In the current implementation, DCONT = 1 can only happen when this is the initial trace message, in which case I-CNT is set to zero.
2	F-ADDR	1-32		Full branch target address
3	TSTAMP	0-64		Time value

Table 15–72. Indirect Branch with Synchronization Message Format for Xtensa LX

Packet	Field	Size (bits)	Value	Description
1	TCODE	6	12	Message type code
	DCONT	1		Discontinuity: 0 = follows another message, 1 = new sequence
	B-TYPE	1		Branch type: 0 = normal branching, 1 = exception or interrupt
	I-CNT	1-10		Number of instruction bytes executed since the last branch or sync. message (correlation messages do not affect I-CNT). Does not include the instruction (pointed to by F-ADDR) causing this branch (B-TYPE=0) or that took the exception/interrupt and thus did not commit (B-TYPE=1). If DCONT is 1, this field is undefined by Nexus. In the current implementation, DCONT = 1 can only happen when this is the initial trace message, in which case I-CNT is set to zero.
2	F-ADDR	1-32		Full branch target address
3	TSTAMP	0-64		Time value

If the example code sequence of Section 15.2.1 results in an indirect branch with synchronization message for the CALL0 instruction, it is constructed as follows:

- DCONT = 0 (trace capture started before the code sequence)
- B-TYPE = 0 (not an exception or interrupt)
- I-CNT = 6 (total size of add and s16i instructions)
- F-ADDR = 0x150 (call target address)
- TSTAMP = DebugExtTime (Optional current time value)

15.2.3 Synchronization Message

The synchronization message, described in Table 15–73 (for Xtensa NX Processors) and Table 15–74 (for Xtensa LX processors), reports the full program counter and its distance from the last trace message. It allows trace decompression software to reconstruct full program flow addresses from the relative addresses provided in other trace messages, within a given trace fragment.

Specifically, when Status.TRACT is set, TRAX emits a synchronization message:

- At the start of the trace capture session, with DCONT=1. This may be replaced with an indirect branch with synchronization message, as described in Section 15.2.1.
- At the end of the trace capture session. This is always a synchronization message, with DCONT=0.
- Periodically with DCONT=0, if so enabled, according to the SMPER field of the Control register described in Section 15.1.2 on page 250. This may be replaced with an indirect branch with synchronization message, as described in Section 15.2.1.
- With DCONT=0, when the instruction counter (I-CNT) nears overflow of its 10-bit range.
- With DCONT=1, after loss of trace due to buffer overflow; that is, when restarting trace after output to TraceRAM or ATB did not keep up with trace bandwidth.

Table 15–73. Synchronization Message Format for Xtensa NX Processors

Packet	Field	Size (bits)	Value	Description
1	TCODE	6	13	Message type code
	DCONT	1		Discontinuity: 0 = follows another message, 1 = new sequence
	I-CNT	1-11		Number of instruction bytes executed since the last branch or sync. message (correlation messages do not affect I-CNT). Does not include the next instruction, pointed to by PC. This field is normally 10 bits wide, with an 11th bit (msbit, bit 10) that is always zero. If DCONT is 1, this field is undefined by Nexus. It is currently implemented as follows (TRAX 3.1 and later). In the initial trace message, it is set to zero. In other messages, it follows (and indicates) a trace overflow condition, in which case: the 11th bit is set if a correlation message was lost (stop trigger occurred) during the overflow condition; and the other 10 bits indicate the number of dropped instruction bytes (however, because I-CNT wraps around in this case when more instructions are dropped than can be represented in 10 bits, the value reported is not necessarily useful).
2	PC	1-32		Full program counter
3	TSTAMP	0-64		Time value

Table 15–74. Synchronization Message Format for Xtensa LX Processors

Packet	Field	Size (bits)	Value	Description
1	TCODE	6	9	Message type code
	DCONT	1		Discontinuity: 0 = follows another message, 1 = new sequence

Packet	Field	Size (bits)	Value	Description
	I-CNT	1-11		Number of instruction bytes executed since the last branch or sync. message (correlation messages do not affect I-CNT). Does not include the next instruction, pointed to by PC. This field is normally 10 bits wide, with an 11th bit (msbit, bit 10) that is always zero. If DCONT is 1, this field is undefined by Nexus. It is currently implemented as follows (TRAX 3.1 and later). In the initial trace message, it is set to zero. In other messages, it follows (and indicates) a trace overflow condition, in which case: the 11th bit is set if a correlation message was lost (stop trigger occurred) during the overflow condition; and the other 10 bits indicate the number of dropped instruction bytes (however, because I-CNT wraps around in this case when more instructions are dropped than can be represented in 10 bits, the value reported is not necessarily useful).
2	PC	1-32		Full program counter
3	TSTAMP	0-64		Time value

The DCONT bit is set to start a new set of trace messages, not contiguous with any previous trace message. It thus reports any discontinuity in the trace message stream. TRAX sets this bit in the first message of a trace capture session, as well as following any period of loss trace due to buffer overflow. Note that TRAX is not expected to encounter an overflow condition if TraceRAM (and ATB, if enabled) always accept data (no busy response): in the current implementation, one 32-bit trace word per cycle is enough to keep up with worst-case processor branching behavior, including single-instruction zero-overhead loop.

For example, given the following instruction sequence executed from address 0x200:

```
0x200:  j     0x220
 :
0x220:  add   a2, a3, a4
0x223:  s16i  a2, a5, 0
0x226:  addi  a3, a3, 4
```

If a synchronization message is emitted when the processor is executing the instruction at address 0x226, it is constructed with:

- DCONT = 0 (trace capture started before the code sequence)
- I-CNT = 6 (total size of add and s16i instructions)
- PC = 0x226 (full program counter of addi instruction)
- TSTAMP = DebugExtTime (Optional current time value)

Note: This above denotes the current TRAX implementation. The standard allows this synchronization message to be constructed as if it occurred *after* the ADDI instruction; that is, with I-CNT = 9 and PC = 0x229.

15.2.4 Correlation Message

This message indicates a trigger occurred, correlating the time of its occurrence to the target's program flow.

Firing of the stop trigger causes a correlation message with EVCODE = 0b001010. Note that TRAX adds to EVCODE two (significant) bits more than the four embodied in Nexus Table 5-21. This is to allow more information regarding trigger cause to be output in future TRAX releases. Future releases will keep the values themselves the same as those recommended by Nexus.

If trace stop was triggered by a PC match, the correlation message normally corresponds exactly to the matching instruction. However, if the matching instruction is a taken branch of any type, the correlation message is output immediately following the branch (or branch and sync) message, in which case the correlation message effectively points to the instruction *following* (in execution order) the one that matched. Also, correlation messages are delayed while the processor executes a single-instruction zero-overhead loop where the instruction takes a single cycle. Thus, in these situations, the instruction at which the trigger occurred is not reported accurately.

Countdown causes a final sync message. They cannot both happen at the same time because of the way that DELAYCOUNT is defined. The stop trigger correlation message has no full PC, but does have I-CNT. Because the final sync message has the full PC, it's always possible to reconstruct backwards, and know the full PC of the trigger point.

Table 15–75. Correlation Message Format

Packet	Field	Size (bits)	Value	Description
1	TCODE	6	33	Message type code
	EVCODE	6	0x0A	Event code
	I-CNT	1-10		Number of instruction bytes executed since the last branch or sync. message (correlation messages do not affect I-CNT).
2	TSTAMP	0-64		Time value

15.2.5 Miscellaneous Information

When the processor is stopped under OCD control, instructions fed to the processor through OCD are counted by the I-CNT field. These instructions are provided by the Xtensa OCD Daemon and are not available from the program executable otherwise run by the processor. As long as the decoding tool (such as `xt-traxview`) knows whether OCD was enabled while tracing, it can detect debug exceptions and treat differently all instructions that follow the exception until return from the exception. This return is usually indicated by an indirect branch message, which corresponds to the RFDO instruction generally used to return to *Running* state (see Section 5.2).

15.2.6 Message Encoding

Each message consists of a series of variable-size packets. Each packet in turn consists of a series of fixed-sized fields (usually only in the first packet of the message) followed by a variable-size field. Thus, each packet can be thought of as a variable-size sequence of bits, with fields starting from the least significant bit, so that the variable-size field ends up in the most significant bits. Each such packet is emitted using a whole number of AUX port transfers. If there are not enough bits for the last transfer, it is padded with zeros. Trailing transfers containing only zero bits may be omitted, that is, zero-valued upper bits may be trimmed away, thus making variable-size fields variable.

TRAX implements the two-pin $\overline{\text{MSEO}}$ protocol with 6 bits of MDO (data pins). Thus, each AUX port transfer fits in exactly one byte of compressed trace. In each byte, the $\overline{\text{MSEO}}$ pins are encoded in the lower two bits, and the MDO pins in the upper six bits.

Thus each packet consists of one or more bytes. $\overline{\text{MSEO}}$ bits are 00 for all bytes except the last one in each packet. $\overline{\text{MSEO}}$ bits of the last byte are 11 for the last packet of each message, and 01 for other message packets.

For example, an indirect branch message without timestamps (see Table 15–69 for Xtensa NX processors, and Table 15–70 for Xtensa LX processors) with `B-TYPE` = 0, `I-CNT` = 17, and `U-ADDR` = 0x123, would be encoded as depicted in Figure 15-44 for Xtensa NX processors and Figure 15–45 for Xtensa LX processors. This message consists of two packets. The first contains `TCODE`, `B-TYPE`, and `I-CNT`, in that order, and the second contains `U-ADDR`. Logically, the packets are first arranged as streams of bits. Because of the little-endian style encoding, they are more easily visualized right-to-left:

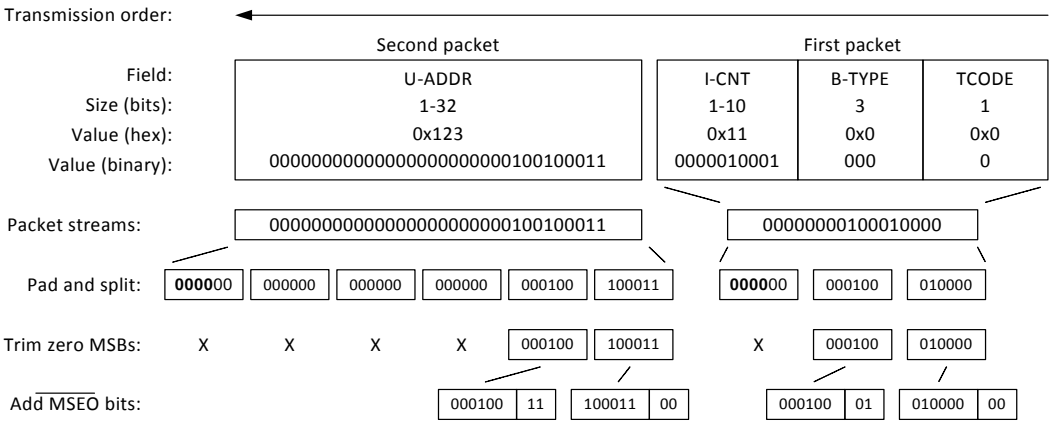


Figure 15-44. Example Message Encoding for Xtensa NX (Indirect Branch without Timestamps)

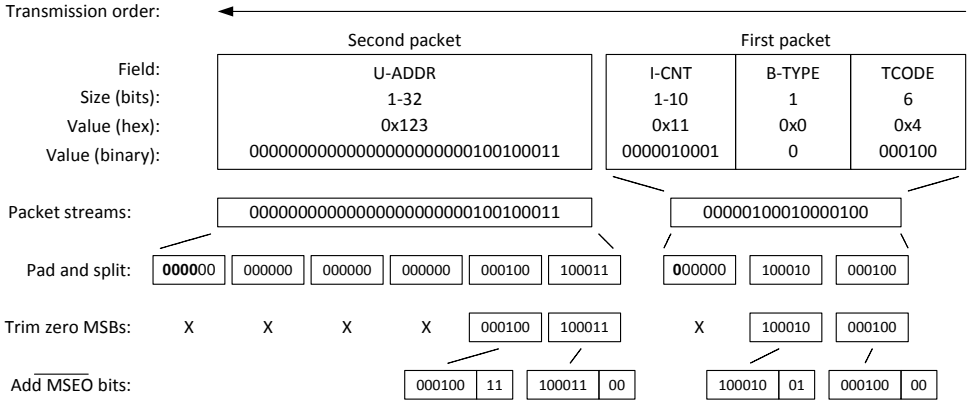


Figure 15–45. Example Message Encoding for Xtensa LX (Indirect Branch without Timestamps)

Once formed into bit streams, packets are transmitted in TDO sized units, that is, in chunks of 6 bits each. The last unit is zero padded if necessary. Any trailing zero-valued units are discarded. However, each packet must have at least one unit, even if its value is zero. Next, $\overline{\text{MSEO}}$ bits are added to demarcate packet and message boundaries.

As shown in Figure 15-44, for Xtensa NX processors the final message consists of the following bytes, in order from left to right: 0x40, 0x11, 0x8c, 0x13. As shown in Figure 15–45, for Xtensa LX processors the final message consists of the following bytes: 0x10, 0x89, 0x8C, 0x13.

15.3 Deviations from the Nexus Standard

For various reasons of simplicity and efficiency, TRAX deviates slightly from the Nexus standard. Specifically, it provides the following optimizations:

- **For Xtensa NX processors only.** The TCODE field width and values have been modified to increase message compression. The Indirect Branch message has significantly more usage in a TRAX message stream than the other messages, and thus has a single bit TCODE (1'b0) for distinguishing it. The Indirect Branch with Synchronization message has a two-bit TCODE (2'b11). The TCODE width of Synchronization and Correlation messages remains 6-bit but the value for Synchronization messages is 6'h0d instead of 6'h09 to be orthogonal from the other changed encodings
- **For Xtensa NX processors only.** The B-TYPE field width has been increased from 1 to 3 bits to allow additional encodings for Loopback (3'b010), OCD Exceptions (3'b110), and OCD Exception Returns (3'b100).
- The I-CNT field reports the number of instruction bytes executed since the last branch or synchronization message, rather than since the last branch message. In other words, the TRAX compressor hardware and decompresser software update their internal instruction counter on every program trace message (other than correlation messages), not just on branch messages. This provides two benefits. First, resource full messages are not needed when the compressor's I-CNT register overflows. Instead, the compressor merely needs to send a synchronization message whenever I-CNT is about to overflow, if it isn't already sending out another branch or synchronization message. This simplifies hardware and helps avoid potential trace overrun conditions. Second, because I-CNT is effectively cleared more frequently, the variable-size I-CNT field is occasionally smaller, reducing the size of the generated trace.
- The Nexus standard mandates at least one synchronization message per 256 messages transmitted. In TRAX, synchronization messages are optional. synchronization messages are required when continuously streaming trace information to an external AUX port, where an external tool may start processing the trace stream at an arbitrary point without stopping the stream. TRAX usually operates without an external AUX port: the entire captured trace goes into a finite size local trace buffer, and tracing stops before reading and processing the trace information. Also, TRAX traces can always be processed backwards in the absence of indirect branch with synchronization messages in the middle of the trace; Such messages can be avoided by turning off periodic synchronization messages. This way, the entire TraceRAM is used, and the overhead of periodic synchronization messages is avoided. Without this feature, if the first synchronization message in the captured trace data happens to be an indirect branch with synchronization message, the portion of the captured trace that precedes this message is unusable. The program counter values prior to that message are unknown, and the message does not provide enough information to reconstruct them.

15.4 TRAX Captured Trace File Format

The `xt-traxcmd` tool (see Chapter 11) writes each captured trace to a file that can be read by the `xt-traxview` tool (see Chapter 12). Other tools may be used to capture traces externally (obtained over the ATB interface). Each trace file consists of one or more headers, shown in Table 15–76, followed by the compressed trace obtained from the TraceRAM or external trace buffer. A trace with more than one header may be generated for a multicore Xtensa system with one header for each core that produces trace data. Future software releases may add more information and sections to this file, so the location of trace data within the file must be obtained from the header.

Table 15–76. Trace File Header

Offset (bytes)	Size (bytes)	Field Name	Description
0x00	8	magic	File type identifier ("TRAXdmp\0")
0x08	1	endianness	0 = little-endian (general use), 1 = big-endian
0x09	1	version	Currently always 1
0x0A	2	*reserved*	*(write as zero, ignore on read)*
0x0C	4	filesize	Size in bytes of entire file, including header
0x10	4	trace_ofs	Start of the TRAX trace data, byte offset from the start of the trace file
0x14	4	trace_size	Size in bytes of trace data
0x18	4	dumptime	Date/time trace was captured (Unix format: seconds since Jan 1st, 1970); 0 if unknown / unavailable
0x1C	4	flags	Miscellaneous flags (see TRAX_FHEADF_***)
0x20	16	username	User that invoked `xt-traxcmd`. Up to 15 chars, null terminated. May be empty if unknown.
0x30	24	toolver	Name and version of tool used to capture/save trace. Up to 23 chars, null terminated.
0x48	40	*reserved*	*(write as zero, ignore on read)*
0x70	8	configid[2]	Processor configuration ID values, 0 if unknown
0x78	8	*reserved*	*(write as zero, ignore on read)*
0x80	32 x 4	trax_regs[]	Values of selected TRAX registers at time of dump, captured after trace has stopped (`TRAXSTAT.TRIG`=1 and `TRAXSTAT.TRACT`=0). Index of array is the TRAX register number. For proper decoding (e.g. detecting wraparound, all registers (0 thru 9) other than the `TRAXDATA` need to be saved here; others may be left zero.
0x100			(Total header size = 256 bytes)

Flags include:

- TRAX_FHEADF_OCD_ENABLED 0x00000001. Set if OCD was enabled while capturing trace (for TRAX 1.0).
- TRAX_FHEADF_TESTDUMP 0x00000002. Set if is a test file (unspecified).
- TRAX_FHEADF_EXTERNAL 0x00000004. Set if trace captured externally, e.g. as obtained over the ATB interface, rather than from TraceRAM. (unimplemented)

The trace data itself must be a contiguous chunk of trace. If captured from TraceRAM, it must be all available trace data. If wrap-around occurred, the oldest data is dumped first, that is: from ADDRESS to end of memory (or to MemEndAddr if present), then from start of memory (or MemStartAddr if present) to just before ADDRESS, so as to obtain a contiguous sequence of trace data.

TRAX trace for a multicore system is a sequence of entries with the following structure:

Table 15–77. TRAX Trace for Multicore System

Offset (bytes)	Size (bytes)	Description
0	1	ATID of the Xtensa core that produced the following ATDATA
1	4	ATDATA

The appearance order of the data entries in the TRAX trace is not defined.

15.5 Trace Session Timeline

Figure 15-46 shows the sequence of events in a typical trace capture session.

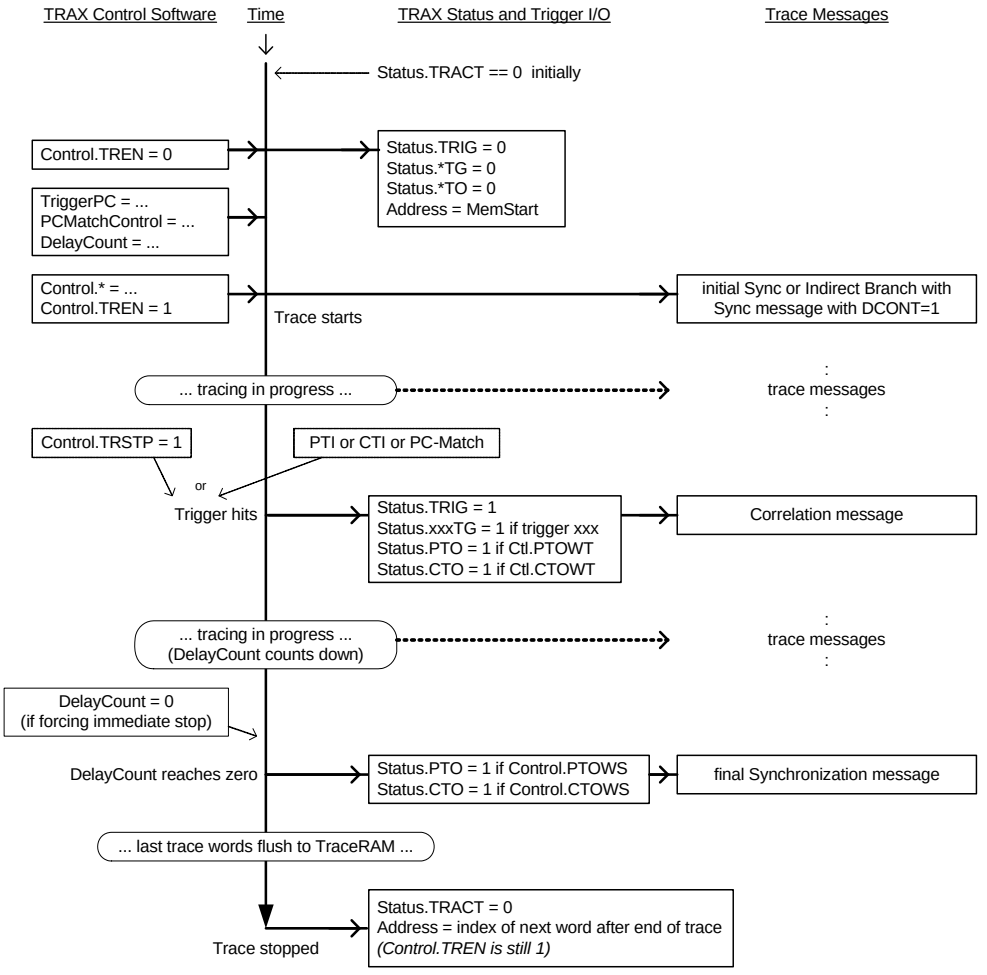


Figure 15-46. Typical Trace Session

The TRAX control software manages the trace capture session by accessing TRAX registers. It first ensures trace is not active (TRAXSTAT.TRACT is zero) before starting, then clears TRAXCTRL.TREN, sets up various registers, and then explicitly starts tracing by writing TRAXCTRL.TREN set to one. It then polls for trace completion by reading TRAXSTAT.TRACT bit until it clears. While waiting for trace completion, it may also poll other registers to report progress, such as stop trigger occurrence (TRAXSTAT.TRIG), trace stream position (TRAXADDR), or delay countdown (DELAYCOUNT). It might also abort the trace by initiating the stop trigger itself (setting TRAXCTRL.TRSTP) and clearing DELAYCOUNT for a practically immediate stop. It can also simply reset the entire Debug module for an instantaneous stop, if the resulting trace is to be discarded and a debug Module reset is appropriate.

Once started, trace normally occurs in three phases: the pre-trigger phase, the post-trigger (delay) phase, and the trace-complete phase. See Figure 15-39 on page 253 for a more detailed state diagram.

TRAX implements a single *reference* or *stop* trigger. It is effectively a stop trigger if DelayCount is initialized to zero. In that case, there is always at least one cycle of delay between emitting the correlation message when the trigger hits, and emitting the final synchronization message and asserting any output triggers.

Note: TRAX only implements a manual *start* trigger. Tracing is always started explicitly by setting the TREN bit of the Control register.

15.6 Tracing from Processor Reset

It is possible to trace from processor reset by starting trace while the core is held in reset (i.e. in hardware by holding `BReset` asserted or in software by setting `CoreReset`) and the Debug module is not (i.e. similarly `DReset` de-asserted or `DebugReset` clear). Alternatively, the processor can be stopped via OCD right after core reset, for example using the GDB `reset` command (assuming RE-2013.2 or later hardware, or the JTAG probe and target board both support control of the system reset line by the Xtensa OCD Daemon), followed by starting trace, then letting the processor resume execution.

15.6.1 Tracing Multiple Processors

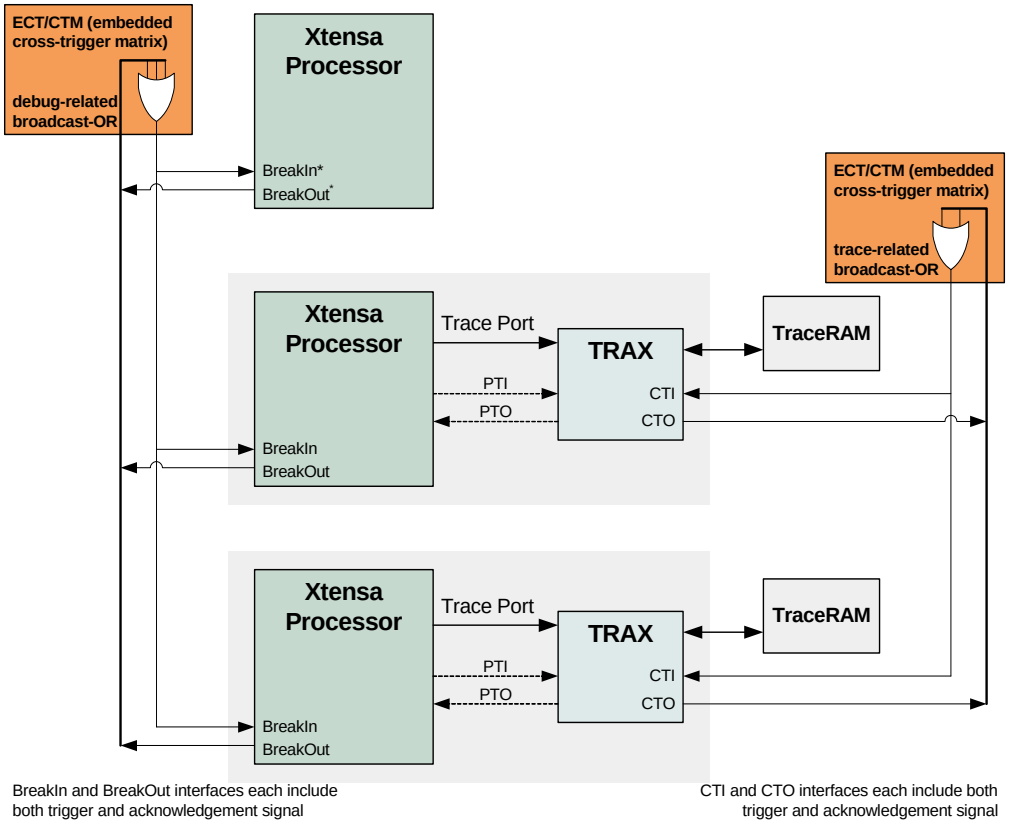


Figure 15-47. TRAX Multi-Processor and Multi-TRAX Network Example

To support trace-to-middle on a *set* of processors, with each processor stopping cleanly after the trace, setup a full cross-trigger with each CTO setup to trigger on stop and CTI enabled, and setup each PTO to trigger on halt (on zero-count) to stop each processor. Adjust the delay counts to take into account the few messages that might get traced between a trigger-out to a processor and the processor actually stopping.

16. Traceport for Xtensa NX Processors

This chapter describes the micro architecture of the Traceport, which is an Xtensa NX processor bus that provides information about its internal state.

16.1 Overview

The Traceport outputs information about the processor's internal state during normal operation. The Traceport enables:

- Program tracing
  - in a software simulation environment
  - in hardware, providing the information required to create compressed program trace (for example, by TRAX)
- Load/store data tracing
- Performance monitoring
  - of events happening in the pipeline e.g. retiring instructions, stalls, bubbles
  - of events that occur independent of the processor pipeline

Information output on the Traceport is approximate; due to the speculative nature of the Xtensa pipeline, and the fact that many events can happen concurrently, it is not possible to reflect (in its entirety) the internal state of the processor.

The Traceport appears as a primary interface of Xtmem. If the Debug module is present (which is required by the TRAX and Performance Monitor options) the Traceport has additional loads inside Xttop.

The basic Traceport signals are present with the Traceport option, and the DataTrace option adds LoadStore-related signals. In addition, other configuration options such as the Performance Monitor, number of LSUs/LSwidth, inbound PIF, prefetch, memory errors, etc. affect the structure of the Traceport viz. cause signals or fields to become present.

16.2 Basic Signals

Table 16–78 lists the signals of the Traceport. The Traceport is geared mostly around the W stage of the processor pipeline and PIF transactions. Described another way, the information that is reported on the Traceport is fundamentally of two categories:

a. Information related to an instruction completing (a.k.a. committing or retiring). Completion happens in the W stage of the pipeline.

b. Information related to a bubble. A bubble cycle is one where no instruction completed in the given W. This could be due to GlobalStall, replay, etc.

The fifth column of Table 16–78 indicates the category to which each of the Traceport signals belongs.

Table 16–78. Traceport Signals

Name	Width	Dir	Config Option	Category
PDebugEnable	1	Input	Trace	n/a
PDebugInst	32	Output	Trace	a
PDebugStatus	8	Output	Trace	a, b
PDebugData	32	Output	Trace	a, b
PDebugPC	32	Output	Trace	a, b
PDebugLS*n*Stat	32	Output	Performance Monitor OR DataTrace	a, b
PDebugLS*n*Addr	32	Output	DataTrace	a, b
PDebugLS*n*Data	32	Output	DataTrace	a, b

The Dir column of Table 16–78 lists the direction of the signals with respect to Xtensa. In other words, the Traceport is exported from Xtmem, and Dir applies with respect to Xtmem.

The hardware cost of the Traceport is dependent upon the configuration options selected.

16.2.1 PDebugEnable

The `PDebugEnable` signal is used to save power in configurations with the Traceport and functional clock gating. When the Traceport output from the Xtensa core is to be consumed, as in an active debug session, `PDebugEnable` must be set high. At other times, `PDebugEnable` can be set low. If the configuration has functional clock gating, setting `PDebugEnable` low gates off clocks to the flip-flops in the Traceport logic. This eliminates toggles in the Traceport logic and in the Xtensa Traceport interface, preventing wasted power.

PDebugEnable is used to gate the clock to the flip-flops in the Traceport logic. Due to the pipelining of trace data within the processor, the Traceport output will be undefined for several cycles after PDebugEnable is asserted. In the cycles where data is undefined, In the cycles where data is undefined, PDebugStatus[5:0] will be "1111_00", to indicate "All other pipeline bubbles".

PDebugEnable can be changed dynamically while the processor is running. It is the designer's responsibility to control this interface using logic external to the processor. The signal can be tied high to always enable trace output, tied low to prevent any trace output, or controlled by external logic that can dynamically change the value.

There are internal users of the Traceport—such as TRAX and the Performance Monitor—which also assert the PDebugEnable signal when the functions (i.e. PC tracing or performance monitoring) are enabled. This is achieved via an OR gate inside Xtensa that combines the respective PDebugEnable signals.

16.2.2 Instruction Completion

Instructions normally complete in the W stage. The size of the instruction in the W is given by PDebugInst[7:0] in number of bytes. PDebugInst[7:0] is 8'b0 if in a given cycle the W stage has no completing instruction. This is also known as a bubble cycle.

The other fields of PDebugInst are detailed in Table 16–86 below.

16.3 Pipeline Bubbles

As mentioned in Section 16.2.2, PDebugInst[7:0] is 8'b0 if in a given cycle the W stage has no completing instruction. The cause of the bubble is reported on PDebugStatus[5:0] as shown in Table 16–79. PDebugStatus[7:6] is reserved.

Table 16–79. Pipeline Bubbles

PDebugStatus [5:0]	Bubble Category	PDebugData	PDebugPC
0000_00	Power shut off	32'b0	32'b0
0000_10	Register dependency or resource conflict	undef	undef
0001_00	Control transfer bubble	undef	undef
0001_01	Branch mispredict	undef	undef
0011_00	L1 scalar D-cache miss (excl uncached miss)	see Table 16–80	undef
0011_10	L1 vector D-cache miss (excl uncached miss)	see Table 16–80	undef

Table 16–79. Pipeline Bubbles (continued)

PDebugStatus [5:0]	Bubble Category	PDebugData	PDebugPC
0100_00	Exception or interrupt (W stage)	Vector address	undef
0100_01	Exception or interrupt (W+1 stage)	see Table 16–81	Exception PC
0101_00	Instruction replay (other)	inst VA	undef
0110_00	HW ITLB refill	undef	undef
0110_10	ITLB miss	undef	undef
0111_00	HW DTLB refill	undef	undef
0111_10	DTLB miss (1-LS config only)	undef	undef
1000_00	Stall	see Table 16–82	undef
1010_00	WAITI mode	undef	undef
1011_00 to 1110_11	reserved	undef	undef
1111_00	all other bubbles	undef	undef

16.3.1 Dependency Information

PDebugStatus[5:0] = 0000_10 indicates a pipeline bubble caused by a pipeline or resource dependency. Dependency examples are register, cache (due to IHI/III/LICT/LICW/SICT/SICW instructions), and multicycle (resulting in R3 replays).

16.3.2 Control Transfer Bubble

PDebugStatus[5:0] = 0001_00 signals the latency cycles attendant to any non-linear change in PC e.g. branch, jump, call, return, loop, exception, etc.

16.3.3 Branch Mispredict

PDebugStatus[5:0] = 0001_01 signals the latency cycles attendant to a mispredicted branch in the W cycle that it would have retired.

16.3.4 L1 Scalar Data Data Cache Miss

PDebugStatus[5:0] = 0011_00 indicates a pipeline bubble upon a D-cache miss from the scalar pipeline.

The type of miss is shown in Table 16–80. This is a subset of the information available on PDebugLS*n*Stat described in Table 16–87. Specifically, “Miss” on PDebugData[4] is as described in Section 16.6.1.4.

Uncached load/store information is not reported with the basic Traceport. The customer has to select the DataTrace or Performance Monitor option to see uncached load/store information. See “Uncached” on PDebugLSnStat[12:11] in Section 16.6.1.5.

16.3.4.1 How to Calculate L1S D-cache Miss Rate

Given that uncached loads and stores constitute a very small number of the total memory accesses, the cache miss rate can simply be calculated by dividing the number of misses (number of cycles PDebugStatus[5:0] = 0011_00) by the total number of loads and stores.

For configurations without DataTrace or using the Performance Monitor, it is necessary to do a static calculation of the scalar loads and stores in the software before it is run on the processor. The miss rate would then be the number of L1S D-cache misses measured from the Traceport divided by that static count.

Scalar loads and stores to local D memories would generally be regarded as hits. In configurations without DataTrace or using the Performance Monitor, it is not possible to know the hit rate specific to the L1S D-cache alone.

For configurations with DataTrace or using the Performance Monitor, the total number of scalar loads and stores, the number of L1S D-cache misses and the target of the scalar load/store are all individually measurable from the Traceport.

Table 16–80. Data Cache Miss

Field of PDebugData	Description	Encoding
PDebugData[31:24]	Reserved	
PDebugData[23:16]	Reserved	
PDebugData[15:8]	LS1 D-cache miss information	same as PDebugData[7:0]
PDebugData[7:5]	reserved (for LS0)	
PDebugData[4]	L1S or L1V D-cache miss (LS0)	0: not a miss 1: miss
PDebugData[3:2]	reserved (for LS0)	
PDebugData[1:0]	Type of load/store instruction in LS0	00: neither 01: load 10: store 11: other

16.3.5 L1 Vector Data Data Cache Miss

PDebugStatus[5:0] = 0011_10 indicates a pipeline bubble upon a D-cache miss from the vector pipeline (as opposed to the scalar pipeline described inSection 16.3.4).

The type of miss is shown in Table 16–80. This is a subset of the information available on PDebugLS*n*Stat described in Table 16–87. Specifically, “Miss” on PDebugData[4] is as described in Section 16.6.1.4.

16.3.5.1 How to Calculate L1S D-cache Miss Rate

The L1V D-cache miss rate can be calculated similar to the description for the L1S D-cache in Section 16.3.4.1, with one complication. The total memory accesses needs to exclude the accesses that hit in the L1S D-cache. This means the cache miss rate can be calculated by dividing the number of misses (number of cycles PDebugStatus[5:0] = 0011_10) by the total number of loads and stores minus the number of L1S D-cache hits (number of cycles PDebugLSnStat[10] is active when PDebugInst[7:0] is NOT 8’h0).

16.3.6 Exceptions and Interrupts

PDebugStatus[5:0] = 6b’0100_00 indicates an instruction in the W stage that took an exception or interrupt. The virtual address of the instruction that was killed appears on PDebugPC[31:0]. The target address of the exception or interrupt vector appears on PDebugData as shown in Table 16–81. It is not possible to count (say) HW refill miss/error unless the Traceport also exports the EXCCAUSE. This is also part of PDebugData.

PDebugStatus[5:0] = 6b’0100_01 indicates an instruction in the W+1 stage that took an exception or interrupt. The target address of the exception or interrupt vector appears on PDebugData[31:0].

Table 16–81. Exception Cause and Encoded Vector Information

Field of PDebugData	Description	Encoding
PDebugData[31:16]	EXCCAUSE	see ISA book for encoding
PDebugData[15:5]	reserved	
PDebugData[4:0]	Exception Vector	0: no vector 1: Reset 6: Double Exception 8: Exception 9: Interrupt 12: Window Overflow 13: Window Underflow 21: Window Over Zero 22: Window Spill 23: Warm Reset

16.3.7 Instruction Fetch Replay

PDebugStatus[5:0] = 6b'0101_00 indicates the first cycle associated with an instruction that should have retired in the W stage but was instead replayed. Note that this is only for instructions that do not fall into any of the other bubble categories. In other words, D-cache miss for instance will result in a replay of the load or store, but in the relevant W, PDebugStatus will indicate D-cache miss and not instruction replay.

When PDebugStatus[5:0] = 6b'0101_00, PDebugData has the PC of the replaying instruction.

16.3.8 Stall Information

PDebugStatus[5:0] = 1000_00 indicates a processor R stage stall (Stall_R). The cause of the stall is further classified in the PDebugData field as shown in Table 16–82.

Table 16–82. Stall Information

Field of PDebugData	Description
PDebugData[31:10]	Reserved
PDebugData[9]	LS related Stalls including L1 dCache misses
PDebugData[8]	Uncached Instruction Fetch Stall
PDebugData[7]	L1 iCache Miss Stall
PDebugData[6]	L1 Vector dCache Miss Stall from LS1 (if LS1 configured)

Table 16–82. Stall Information (continued)

Field of PDebugData	Description
PDebugData[5]	L1 Vector dcache Miss Stall from LS0
PDebugData[4]	L1 Scalar dCache Miss Stall from LS1 (if LS1 configured)
PDebugData[3]	L1 Scalar dCache Miss Stall from LS0
PDebugData[2]	Other resource dependencies than LS, cache and ifetch
PDebugData[1:0]	Reserved

16.3.9 WaitI Mode

PDebugStatus[5:0] = 1010_00 indicates that the processor has entered internal WAITI mode. The external PWaitMode signal will be asserted later than the WAITI encoding seen on Traceport. How many cycles later is not fixed, and depends on transactions in flight on Xtensa interfaces such as TIE ports or AXI.

16.3.10 Other Bubbles

PDebugStatus[5:0] = 1111_00 indicates all other pipeline bubbles. These bubbles are often associated with previous bubble causes.

16.3.11 Special Use of PDebugPC

In certain bubble cases, Xtensa uses PDebugPC for purposes not immediately intuitive. In particular, TRAX requires a valid PC even when there is no valid instruction of exception signaled on the Traceport. It needs this to correctly construct sync messages for instance. To accommodate this the Traceport speculates the PC as follows:

- in the cycle after a branch or exceptOrInt_X, if there is no valid instruction, PDebugPC shows the target PC because that is likely the next retiring PC
- in the cycle after a valid instruction, if there is no valid instruction - PDebugPC shows the sequential PC

16.4 Instruction Types

When PDebugInst[7:0] is not 8’b0, a valid instruction is retiring in the W stage. PDebugStatus[5:0] gives the type of instruction as shown Table 16–83. PDebugStatus[7:6] is reserved.

Table 16–83. Types of Instructions

PDebusStatus [5:0]	Description	PDebugData	PDbg PC
0000_00	JX	Target virtual address	**0000_00**
0001_00	CALLX	Target virtual address	PC
0010_00	All call returns	Target virtual address	PC
0011_00	All exception returns	Target virtual address	PC
0100_00	Branch taken or loop not taken	Target virtual address	PC
0101_00	J	Target virtual address	PC
0110_00	CALL	Target virtual address	PC
0111_00	Branch not taken	*undef*	PC
1000_00	Loop instruction (taken)	*undef*	PC
1001_00	S32EX/L32EX	Conditional store data	PC
1010_00	WSR/XSR to LBEGIN	SR Write data value	PC
1011_00	WSR to MMID	SR Write data value	PC
1100_00	RSR or WSR (except MMID and LBEGIN), or XSR (except LBEGIN), or S32E (when an RSR is issued as part of it), or LDDR32.P or SDDR32.P	SR number and inst type (see Table 16–84)	PC
1101_00	RER or WER	ER address and inst type (see Table 16–85)	PC
1110_xx	Reserved	*undef*	*undef*
1111_00	Default (none of the above)	*undef*	PC

16.4.1 Instruction Virtual Address

The PC interface, PDebugPC[31:0], indicates the value of the program counter when an instruction completes in the Xtensa processor pipeline; as indicated by PDebugInst[7:0] not equal to 8'h0. When PDebugInst[7:0] is 8'b0, this interface is undefined, except for exceptions/interrupts when PDebugStatus[5:0] = 010000.

16.4.2 Control Transfer Instructions

PDebugStatus[5:0] = 0100_00 denotes a taken branch, and PDebugStatus[5:0] = 0111_00 denotes a branch not taken.

PDebugStatus[5:0] = 0100_00 also indicates a loop instruction where the loop is not taken. Loop instructions LOOPGTZ and LOOPNEZ behave like a branch if the loop condition is not satisfied. PDebugStatus[5:0] = 1000_00 indicates a taken loop and PDebugStatus[5:0] = 1010_00 indicates writes to the LBEGIN Special Register.

Note: When in OCD mode, all Control Transfer Instructions will function as branches that are not taken, and thus will have that encoding on PDebugStatus[5:0] (0111_10).

16.4.3 S32EX/L32EX

1001_00 is output on PDebugStatus[5:0] for a `S32EX` or `L32EX` instruction.

The address and load data are available on the LS signals as shown in Table 16–88.

16.4.4 Special Registers

1010_00 is output on PDebugStatus[5:0] for a `WSR.LBEG` instruction, 1011_00 is output on PDebugStatus[5:0] for a `WSR.MMID` instruction, and 1100_00 is output for writes to all other special registers.

MMID, (memory map ID) is a 32-bit wide write-only special register (SR number 89) that is configured only when the Traceport option is selected. The MMID register contents are reflected on the Traceport to help in decoding the trace output by defining which memory map is in force.

1010_00 is output on PDebugStatus[5:0] for an `XSR.LBEG` instruction, and 1100_00 is output for XSR instructions directed at all other special registers.

When PDebugStatus[5:0] = 1100_00, PDebugData indicates the SR number and instruction type as shown in Table 16–84 below. Also, the RSR/WSR/XSR read/write data are available on the LS signals as shown in Table 16–88.

Table 16–84. SR Number and Instruction Type

Field of PDebugData	Description
PDebugData[31:11]	Reserved
PDebugData[10]	XSR Instruction
PDebugData[9]	WSR Instruction
PDebugData[8]	RSR Instruction
PDebugData[7:0]	Special Register Number

16.4.5 External Registers

When PDebugStatus[5:0] = 1101_00, PDebugData indicates the ER address and instruction type as shown in Table 16–85. External registers are 32 bits wide, and addressed using a 30-bit address, (i.e. the least significant two bits of the AR are don't-care) therefore the complete address is potentially available.

Table 16–85. ER Address and Instruction Type

Field of PDebugData	Description
PDebugData[13:2]	ER address
PDebugData[1]	WER address
PDebugData[0]	RER Instruction

16.4.6 Other Instructions

PDebugStatus[5:0] = 1111_00 when none of the other PDebugStatus encodings apply.

16.5 Instruction Status

PDebugInst gives the instruction or fetch status of each retiring instruction. A pipeline bubble is indicated by PDebugInst[7:0] = 8'b0. The other fields of PDebugInst are valid only when PDebugInst[7:0] is non-zero.

Table 16–86. Instruction Status

Field of PDebugInst	Description	Encoding
PDebugInst [7:0]	Instruction size	number of bytes, e.g. 8'b0000_0110 if 48-bit wide instruction
PDebugInst [8]	Reserved	
PDebugInst [11:9]	Reserved	
PDebugInst[14:12]	Instruction source (field present only when config has >1 local inst mem)	000: reserved or not implemented 001: AXI 010: I-cache 011: InstRAM0 100: InstRAM1 101: Reserved 110: Reserved 111: OCD
PDebugInst [15]	Reserved	

Table 16–86. Instruction Status (continued)

Field of PDebugInst	Description	Encoding
PDebugInst[16]	Reserved	
PDebugInst [19:17]	ExecLevel	000: Reserved 001: Reserved (User/Pageable code) 010: Application code 011: Exception handler 100: Interrupt handler 101: Non-interruptible dispatch code 110: Reserved 111: Reserved
PDebugInst[23:20]	Loopback status [23] Reserved [22] Reserved [21] last inst of loop [20] loopback will occur	0x0x: not the last inst of either loop xx10: last inst of loop, no loopback x011: last inst of loop, w loopback
PDebugInst [31:24]	Highest Active Interrupt priority	Interrupt Priority Number

16.5.1 Instruction Size

The size of the instruction in the W stage is given by PDebugInst[7:0]. The size is simply expressed in bytes, e.g. 8'b0000_0011 if the instruction is 24-bit wide. PDebugInst[7:0] is 8'b0 if in a given cycle the W stage has no completing instruction.

16.5.2 Instruction Source

PDebugInst[14:12] gives the source of the instruction. This field is present only in configurations with more than one local instruction memory e.g. two instruction RAMs, InstRAM0 and I-cache, etc. For the purposes here, an I-cache is considered a local memory. In configurations with only one local instruction memory, the field has value 3'b0.

The information in PDebugInst[14:12] can for instance be used to count fetches from IRAM0—as opposed to other sources. It's possible that an instruction comes from two sources—when an instruction straddles an IRAM boundary for instance. In such cases PDebugInst[14:12] refers to the trailing end of the instruction.

PDebugInst[14:12] = 001 means that the instruction came from an uncached AXI fetch. Uncached AXI transactions are countable through the information provided on PDebug-Status as described in section Table 16.6.1.5. However, the two numbers would not match because the former corresponds to retiring instructions whereas the latter corresponds to uncached AXI transactions.

PDebugInst[14:12] = 010 means that the instruction was supposed to hit in the I-cache, and it did.

I-cache miss information is available from the stall bubble information of Table 16–82 corresponding to the killed W. Upon the replay—and assuming that the instruction actually hits this time and retires properly—Hit will be true in the W-stage.

If both Miss and Hit are false, look elsewhere for information regarding the killed or retiring instruction.

PDebugInst[14:12] = 111 signals that the instruction is from OCD, which could be used by agents such as TRAX.

16.5.3 Loopback Status

For every valid instruction i.e. PDebugInst[7:0] not 0, PDebugInst[23:20] provides loopback information.

In the case where the last instruction of a loop is a branch, LCOUNT is undefined and this branch must be used to exit the loop. Because LCOUNT is undefined, the Traceport hardware will indicate a taken branch on PDebugStatus.

16.6 Load/Store Instructions

For each load-store unit configured, selecting the DataTrace option gives three 32-bit signals, PDebugLS*n*Stat, PDebugLS*n*Addr and PDebugLS*n*Data.

16.6.1 Load/Store Status

As shown in Table 16–87, PDebugLS*n*Stat gives detailed information about the ld/st operation when there is a retiring load or store in the W stage. In this case, PDebugLS*n*-Stat[3:0] will have a non-zero value. In the following cases, but other PDebugLS*n*-Stat[3:0] will be zero but other PDebugLS*n*Stat fields will contain valid information

- when there is a killed load or store in the W stage, if there is a LS reason for the kill, such as a L1 scalar or vector D-cache miss
- when there is a HW-assisted TLB refill instruction in the W-stage

This characteristic is different from PDebugInst, whose fields (other than the instruction size PDebugInst[7:0]) are valid only when there is a valid instruction retiring in the W stage.

The fields of PDebugLS*n*Stat are not valid:

- when there is a non-ld/st instruction in the W stage, regardless retiring or killed
- when there is a killed load or store in the W stage, if the reason for the kill was not related to LS
- regardless of valid LS operations/functions in other stages of the pipeline e.g. bank-conflict detection, tag lookup, etc.
- when Stall_R is being asserted, even if the original cause of the Stall_R was LS-related (e.g. L1 scalar or vector D-cache miss)

In these cases, PDebugLS*n*Stat[3:0] will be zero.

Table 16–87. Load/Store Information

Field of PDebugData	Description	Encoding
PDebugLS*n*Stat[3:0]	Type of instruction in LS*n*	0000: neither 0001: hw itlb refill 0010: hw dtlb refill 0011: reserved 0100: reserved 0101: load 0110: store 0111: reserved 1000: l32r 1001: reserved 1010: l32ex 1011: s32ex 1100: cache test inst 1101: reserved 1110: rsr/wsr/xsr/lddr32.p/sddr32.p 1111: rer/wer
PDebugLS*n*Stat[7:4]	Operand size	0000: 8-bit data 0001: 16-bit data 0010: 32-bit data 0011: 64-bit data 0100: 128-bit data 0101: 256-bit data 0110: 512-bit data 0111 and above: reserved
PDebugLS*n*Stat[8]	Data TLB miss	0: not a miss 1: miss
PDebugLS*n*Stat[10:9]	Reserved	0: not a miss 1: miss
PDebugLS*n*Stat[11]	Uncached Load	0: not uncached 1: uncached
PDebugLS*n*Stat[12]	Uncached Store	0: not uncached 1: uncached
PDebugLS*n*Stat[16:13]	External memory attributes	[13] Cacheable Request [14] Writeback region [15 Read-Allocate region [16] Write-Allocate region

Table 16–87. Load/Store Information (continued)

Field of PDebugData	Description	Encoding
PDebugLS*n*Stat[19:17]	Reserved	
PDebugLS*n*Stat[23:20]	Local target	0000: not to local memory 001x: InstRAM (0/1) 010x: InstROM (0/1) 101x: DataRAM (0/1) 111x: DataROM (0/1) others: reserved
PDebugLS*n*Stat[24]	L1 Scalar dCache load miss	0: not a miss 1: miss
PDebugLS*n*Stat[25]	L1 Vector dCache load miss	0: not a miss 1: miss
PDebugLS*n*Stat[26]	L1 Scalar dCache store miss	0: not a miss 1: miss
PDebugLS*n*Stat[27]	L1 Vector dCache store miss	0: not a miss 1: miss
PDebugLS*n*Stat[28]	L1 Scalar dCache load hit	0: not a hit 1: hit
PDebugLS*n*Stat[29]	L1 Vector dCache load hit	0: not a hit 1: hit
PDebugLS*n*Stat[30]	L1 Scalar dCache store hit	0: not a hit 1: hit
PDebugLS*n*Stat[31]	L1 Vector dCache store hit	0: not a hit 1: hit

16.6.1.1 Type of LS Instruction

PDebugLS*n*Stat[3:0] shows the type of LS instruction. The information appears for both ld/st that retire or are killed in the W stage.

16.6.1.2 Size

For TIE load or store instructions that use arbitrary byte enables, PDebugLS*n*Stat[7:4] will not indicate the actual number of bytes accessed. For example, if the instruction semantic uses the 4-byte load interface and sets "1100" for the Byte Disable interface, PDebugLS*n*Stat[7:4] will indicate "0010" and not "0001". There is one exception to this rule—if the load or store is entirely killed because the Byte Disable interface is set to "0000," PDebugLS*n*Stat[3:0] will indicate "0000".

16.6.1.3 TLB Miss

PDebugLS*n*Stat[8] reports DTLB misses. DTLB hit information is derivable from the number of successfully retiring ld/st instructions minus DTLB miss information. No translation—i.e. going through one of the DTLB's no-translation/identity-mapped ways—is still regarded as a hit if the ld/st retired.

This bit is absent/undefined in configurations without full MMU.

16.6.1.4 Miss and Hit

PDebugLS*n*Stat[26,24] indicates L1 Scalar D-cache misses (load and store misses are delineated separately) and PDebugLS*n*Stat[30,28] indicates L1 Scalar D-cache hits. PDebugLS*n*Stat[27,25] indicates L1 Vector D-cache misses and PDebugLS*n*Stat[31,29] indicates L1 Vector D-cache hits. In this context, Miss is: load/store was supposed to hit in the cache, but it did not. Note that this is the same information as reported by (for LS0 for instance) PDebugData[4] (Table 16–80) when a L1S D-cache miss bubble is indicated by PDebugStatus[5:0] = 0011_00 (Table 16–79) or a L1V D-cache miss bubble is indicated by PDebugStatus[5:0] = 0011_10 (Table 16–79). Similarly, Hit is defined: load/store was supposed to hit in the cache, and it did. The Miss bit will be true in the W stage of the load or store — again that was supposed to hit in the cache — that was killed. Upon the replay — and assuming that the load/store actually hits this time and retires properly — the Hit bit will be true in the W-stage.

If both Miss and Hit are false, look elsewhere for information regarding the killed or retiring load/store.

Miss and Hit cannot both be true in the same cycle.

To count the true number of cache hits, a user must subtract the misses from the hits. This can be inaccurate in that the replaying load/store could be killed — for example, because of an interrupt — but this effect does not grossly distort the miss/hit rate calculation.

16.6.1.5 Uncached

PDebugLS*n*Stat[11] indicates an uncached load to be issued on the AXI.

PDebugLS*n*Stat[12] indicates an uncached store.

Uncached is used to distinguish local RAM accesses from AXI accesses. Also, it's used to distinguish "No Allocate" from "Allocate" accesses.

16.6.1.6 Local Target

PDebugLS*n*Stat[23:20] signals that the ld/st target is a local memory. This information can for instance be used to count load/stores to DataRAM0 — as opposed to other targets.

Note that local memory accesses are derivable from the address available on PDebugLS*n*Addr, but we have this information anyway for redundancy and ease of counting (i.e. no need for comparator) by external hardware.

16.6.2 Load/Store Address and Data

As shown in Table 16–88, PDebugLS*n*Addr and PDebugLS*n*Data provide the address and data of the operation in the load/store unit. It might be necessary to use a combination of PDebugData and PDebugLS*n*Stat to distinguish the classes of operations listed in the table. Cache Access instructions includes DPFL. The cache address value that appears on PDebugLS*n*Addr for these instructions is the data value of the AR register used by that cache-test instruction.

PDebugLS*n*Data is always fixed at 32 bits, regardless of the true LSwidth. For loads and stores wider than 32 bits, only the least significant 32 bits are available. “Least significant” here refers to the post-rotated MemDataIn or the pre-rotated MemDataOut. In other words, when the data goes to, or comes from, an AR, the 32 bits of the AR will be available on PDebugLS*n*Data regardless of LSwidth, or memory endianness.

Table 16–88. Load/Store Address and Data

PDebugData and PDebugLSnStat	PDebug LS*n*Addr	PDebug LS*n*Data
Miss	Data virtual address	undef
TLB Refill (both I and D)	Load virtual address	TLB write data
Load	Data virtual address	Load data (or 32 least significant bits thereof)
Store	Data virtual address	Store data (or 32 least significant bits thereof)
S32EX/L32EX	Data virtual address	Store/Load data
Cache Access Insts	Cache address	Cache ld or st data (depends upon op)
RSR / XSR / WSR	XSR write data value WSR write data value	RSR read data value XSR read data value
RER / WER	WER write data value	RER read data value
None of the above	*undef*	*undef*

17. Traceport for Xtensa LX Processors

This chapter describes the micro architecture of the Traceport, which is an Xtensa bus that provides information about its internal state.

17.1 Overview

The Traceport outputs information about the processor's internal state during normal operation. The Traceport enables:

- program tracing
  - in a software simulation environment
  - in hardware, providing the information required to create compressed program trace by, for instance, TRAX
- load/store data tracing
- performance monitoring
  - of events happening in the pipeline e.g. retiring instructions, stalls, bubbles
  - of events that occur independent of the processor pipeline

Information output on the Traceport is approximate; due to the speculative nature of the Xtensa pipeline, and the fact that many events can happen concurrently, it is not possible to reflect -- in its entirety -- the internal state of the processor.

The Traceport appears as a primary interface of Xtmem. If the Debug module is present (which is required by the TRAX and Performance Monitor options) the Traceport has additional loads inside Xttop.

The basic Traceport signals are present with the Traceport option, and the DataTrace option adds LoadStore-related signals. In addition, other configuration options such as the Performance Monitor, number of LoadStore units, load/store width, inbound PIF, prefetch, memory errors, etc. affect the structure of the Traceport (specifically, cause signals or fields to become present).

17.2 Basic Signals

Table 17–89 lists the signals of the Traceport. The Traceport is geared mostly around the W stage of the processor pipeline and PIF transactions. Described another way, the information that is reported on the Traceport is fundamentally of five categories:

a. Information related to an instruction completing (a.k.a. committing, a.k.a. retiring). Completion happens in the W stage of the pipeline.

b. Information related to a bubble. A bubble cycle is one where no instruction completed in the given W. This could be due to GlobalStall, replay, etc.

c. Information related to a PIF transaction. This is reported when the transaction is signaled valid on the PIF bus.

d. Information related to prefetch events. The information is generated in the cycle that the prefetch event happened, and is asynchronous/independent of other events such as pipeline or PIF events.

e. Information related to iDMA events. The information is generated in the cycle that the iDMA event happened, and is asynchronous/independent of other events such as pipeline or PIF events.

The fifth column of the following table indicates the category to which each of the Traceport signals belongs.

Table 17–89. Traceport Signals

Name	Width	Dir	Config Option	Category
PDebugEnable	1	Input	Trace	n/a
PDebugInst	32	Output	Trace	a
PDebugStatus	8	Output	Trace	a, b
PDebugData	32	Output	Trace	a, b
PDebugPC	32	Output	Trace	a, b
PDebugLS*n*Stat	32	Output	Performance Monitor OR DataTrace	a, b
PDebugLS*n*Addr	32	Output	DataTrace	a, b
PDebugLS*n*Data	32	Output	DataTrace	a, b
PDebugOutPIF	8	Output	Trace AND {WriteBack Cache OR Prefetch} AND Performance Monitor	c
PDebugInbPIF	8	Output	Trace AND InboundPIF AND Performance Monitor	c

Table 17–89. Traceport Signals (continued)

Name	Width	Dir	Config Option	Category
PDebugPrefetchLookup	8	Output	Trace AND Prefetch AND Performance Monitor	d
PDebugPrefetchL1Fill	4	Output	Trace AND Prefetch AND Performance Monitor	d
PDebugiDMA	8	Output	Trace AND iDMA AND Performance Monitor	e

The Dir column of Table 17–89 lists the direction of the signals with respect to Xtensa. In other words, the Traceport is exported from Xtmem, and Dir applies with respect to Xtmem.

The hardware cost of the Traceport is dependent upon the configuration options selected.

17.2.1 PDebugEnable

The `PDebugEnable` signal is used to save power in configurations with the Traceport and functional clock gating. When the Traceport output from the Xtensa core is to be consumed, as in an active debug session, `PDebugEnable` must be set high. At other times, `PDebugEnable` can be set low. If the configuration has functional clock gating, setting `PDebugEnable` low gates off clocks to the flip-flops in the Traceport logic. This eliminates toggles in the Traceport logic and in the Xtensa Traceport interface, preventing wasted power.

`PDebugEnable` is registered once inside the Xtensa core and the registered version of the signal is used to gate the clock to the flip-flops in the Traceport logic. Due to the pipelining of trace data within the processor, the Traceport output will be undefined for several cycles after `PDebugEnable` is asserted. In the cycles where data is undefined, `PDebugStatus[5:0]` will be "1111_00," to indicate "All other pipeline bubbles."

`PDebugEnable` can be changed dynamically while the processor is running. It is the designer's responsibility to control this interface using logic external to the processor. The signal can be tied high to always enable trace output, tied low to prevent any trace output, or controlled by external logic that can dynamically change the value.

There are internal users of the Traceport -- such as TRAX and the Performance Monitor -- which also assert the `PDebugEnable` signal when the functions (i.e. PC tracing or performance monitoring) are enabled. This is achieved via an OR gate inside Xtensa that combines the respective `PDebugEnable` signals.

17.2.2 Instruction Completion

Instructions normally complete in the W stage. The size of the instruction in the W is given by PDebugInst[7:0] in number of bytes. PDebugInst[7:0] is 8'b0 if in a given cycle the W stage has no completing instruction. This is also known as a bubble cycle.

The other fields of PDebugInst are detailed in Table 17–98 below.

17.3 Pipeline Bubbles

As mentioned above, PDebugInst[7:0] is 8'b0 if in a given cycle the W stage has no completing instruction. The cause of the bubble is reported on PDebugStatus[5:0] as shown in Table 17–90. PDebugStatus[7:6] is reserved.

Table 17–90. Pipeline Bubbles

PDebugStatus [5:0]	Bubble Category	PDebugData	PDebugPC
0000_00	Power shut off	32'b0	32'b0
0000_10	Register dependency or resource conflict	see Table 17–91	undef
0001_00	Control transfer bubble	undef	undef
0010_00	I-cache miss (incl uncached miss)	VA of miss	undef
0010_01	Reserved for future indication of Uncached inst fetch	VA of "miss"	undef
0011_00	D-cache miss (excl uncached miss)	see Table 17–92	undef
0100_00	Exception or interrupt (W stage)	see Table 17–93	Exception PC
0100_01	Exception or interrupt (W+1 stage)	Vector address	undef
0101_00	Instruction replay (other)	inst VA	undef
0110_00	HW ITLB refill	undef	undef
0110_10	ITLB miss	undef	undef
0111_00	HW DTLB refill	undef	undef
0111_10	DTLB miss (1-LS config only)	undef	undef
1000_00	Stall	see Table 17–94	undef
1001_00	HW-corrected memory error	undef	undef
1010_00	WAITI mode	undef	undef
1011_00 to 1110_11	reserved	undef	undef
1111_00	all other bubbles	undef	undef

17.3.1 Power Shut Off

PDebugStatus[5:0] = 0000_00 indicates that the processor domain has been powered off. If PSO has been configured for one domain, this encoding is true when the domain i.e. Xtmem has been powered off, but is only visible if Traceport has been configured and is exported from Xtmem i.e. no debug features. If PSO has been configured for three domains, this encoding is true only when the processor domain has been powered off, and says nothing about the power status of the other domains. The power status of the other domains is available through separate Xtmem I/O; specifically PsoDomainOff-Debug and PsoDomainOffMem.

17.3.2 Dependency Information

PDebugStatus[5:0] = 0000_10 indicates a pipeline bubble caused by a pipeline or resource dependency. When this encoding is true, the causes are further classified in PDebugData fields as shown in Table 17–91. Note that the fields are not mutually exclusive; more than one cause could be asserted in a given cycle e.g. register dependency *and* LS instruction dependency.

Table 17–91. Pipeline Dependency

Field of PDebugData	Hold Bubble Info
PDebugData[31:18]	reserved
PDebugData[17]	HALT instruction (TX only)
PDebugData[16]	MEMW, EXTW or EXCW instruction dependency
PDebugData[15:13]	reserved
PDebugData[12]	register dependencies or resource (e.g.TIE ports) conflicts
PDebugData[11]	store release (instruction) dependency
PDebugData[10:9]	reserved
PDebugData[8]	various LoadStore dependencies (miss handling, prefetch, cache access insts, s32c1i, etc)
PDebugData[7:5]	reserved
PDebugData[4]	reserved for various instruction fetch dependencies (I-cache miss, TLB miss processing, IRAM store, loop, etc.)
PDebugData[3:1]	reserved
PDebugData[0]	all other hold dependencies resulting from data or resource dependencies

PDebugData[8] is asserted only in special situations, e.g. for certain cache access instructions. A normal load/store missing in the cache gets a GlobalStall in the M.

17.3.3 Control Transfer Bubble

PDebugStatus[5:0] = 0001_00 signals the latency cycles attendant to any non-linear change in PC e.g. branch, jump, call, return, loop, exception, etc.

17.3.4 Instruction Cache Miss and Uncached Fetch

PDebugStatus[5:0] = 0010_00 indicates a pipeline bubble upon an I-cache miss.

Uncached fetches are treated as cache misses and also reported with PDebugStatus[5:0] = 0010_00.

17.3.4.1 How to Calculate I-cache Miss Rate

Given that uncached fetches constitute a very small number of the total memory accesses, the cache miss rate can simply be calculated by dividing the number of misses by the total number instructions executed. The former is available on PDebugStatus (as described above) and the latter is available on PDebugInst as detailed in Table 17–98.

Instructions fetched from local I memories would generally be regarded as hits, but if it is necessary to know the hit rate specific to the I-cache alone, instruction source information is also available on PDebugInst as detailed in Table 17–98.

17.3.5 Data Cache Miss

PDebugStatus[5:0] = 0011_00 indicates a pipeline bubble upon a D-cache miss.

The virtual address of the miss is not reported.Instead Traceport reports the type of miss as shown in Table 17–92. This is a subset of the information available on PDebugLS*n*-Stat described in Table 17–99. Specifically, “Miss” on PDebugData[4] is as described in Section 17.6.1.4.

Table 17–92. Data Cache Miss

Field of PDebugData	Description	Encoding
PDebugData[31:24]	Reserved	
PDebugData[23:16]	Reserved	
PDebugData[15:8]	LS1 D-cache miss information	same as PDebugData[7:0]
PDebugData[7:5]	reserved (for LS0)	

Table 17–92. Data Cache Miss (continued)

Field of PDebugData	Description	Encoding
PDebugData[4]	D-cache miss (LS0)	0: not a miss 1: miss
PDebugData[3:2]	reserved (for LS0)	
PDebugData[1:0]	Type of load/store instruction in LS0	00: neither 01: load 10: store 11: s32c1i, other

Note that uncached load/store information is not important enough to warrant reporting with the basic Traceport. The customer has to select the DataTrace or Performance Monitor option to see uncached load/store information. See “Uncached” on PDebugData[5] in Section 17.6.1.5.

17.3.5.1 How to Calculate D-cache Miss Rate

Given that uncached loads and stores constitute a very small number of the total memory accesses, the cache miss rate can simply be calculated by dividing the number of misses by the total number of loads and stores.

For configurations without DataTrace or using the Performance Monitor, it is necessary to do a static calculation of the loads and stores in the SW before it is run on the processor. The miss rate would then be the number of D-cache misses measured from the Traceport divided by that static count.

Loads and stores to local D memories would generally be regarded as hits. In configurations without DataTrace or using the Performance Monitor, it is not possible to know the hit rate specific to the D-cache alone.

For configurations with DataTrace or using the Performance Monitor, the total number of loads and stores, the number of D-cache misses and the target of the load/store are all individually measurable from the Traceport.

17.3.6 Exceptions and Interrupts

PDebugStatus[5:0] = 6b’0100_00 indicates an instruction in the W stage that took an exception or interrupt. The virtual address of the instruction that was killed appears on PDebugPC[31:0]. The target address of the exception or interrupt vector appears in an encoded format on PDebugData as shown in Table 17–93. It is not possible to count (say) HW refill miss/error unless the Traceport also exports the EXCCAUSE. This is also part of PDebugData.

PDebugStatus[5:0] = 6b'0100_01 indicates an instruction in the W+1 stage that took an exception or interrupt. The target address of the exception or interrupt vector appears on PDebugData[31:0]. Even though the vector is available in an encoded format in the W stage, some HW clients are not able to translate that to the actual address. This is especially true in the presence of relocatable vectors.

Table 17–93. Exception Cause and Encoded Vector Information

Field of PDebugData	Description	Encoding
PDebugData[31:22]	reserved	
PDebugData[21:16]	EXCCAUSE	see ISA book for encoding
PDebugData[15:5]	reserved	
PDebugData[4:0]	Encoded Exception Vector	0: no vector 1: Reset 2: Debug (repl corresp level "n") 3: NMI (repl corresp level "n") 4: User 5: Kernel 6: Double 7: Memory Error 8: reserved 9: reserved 10: Window Overflow 4 11: Window Underflow 4 12: Window Overflow 8 13: Window Underflow 8 14: Window Overflow 12 15: Window Overflow 12 16: Int Level 2 (n/a if debug/NMI) 17: Int Level 3 (n/a if debug/NMI) 18: Int Level 4 (n/a if debug/NMI) 19: Int Level 5 (n/a if debug/NMI) 20: Int Level 6 (n/a if debug/NMI) 21-31: reserved

Note that for the Debug and NMI vector encodings of Table 17–93, the corresponding interrupt level "n" is unused/made moot i.e. that encoding will never be seen.

17.3.7 Instruction Replay

PDebugStatus[5:0] = 6b'0101_00 indicates that an instruction that should have retired in the W stage was instead replayed. Note that this is only for instructions that do not fall into any of the other bubble categories. In other words, D-cache miss for instance will result in a replay of the load or store, but in the relevant W, PDebugStatus will indicate D-cache miss and not instruction replay.

When PDebugStatus[5:0] = 6b'0101_00, PDebugData has the PC of the replaying instruction.

17.3.8 HW TLB Refill

PDebugStatus[5:0] = 0110_00 and PDebugStatus[5:0] = 0111_00 indicate respectively hardware ITLB and hardware DTLB refill operations and are applicable only if an MMU with hardware refill is configured. The refill operation is automatically inserted by the processor hardware and does not appear in the software program's object code. This is known as the HW-assisted TLB refill using a special pseudo instruction LDPTE.

The refill address and data are available on PDebugLS0Addr and PDebugLS0Data as shown in Table 17–101.

17.3.9 ITLB Miss

PDebugStatus[5:0] = 0110_10 indicates a pipeline bubble upon an ITLB miss. This is flagged only once—in the W of the instruction that would have retired—so it can be used to count ITLB misses.

Note that one row up in the table PDebugStatus[5:0] = 0110_00 indicates HW ITLB refill. This would not be the same count as ITLB miss because not all ways of the ITLB are auto-refill.

ITLB miss indication and HW ITLB refill will (obviously) never be indicated in the same cycle. Let's take a look at some scenarios to make this distinction clearer.

- Miss on a static way. The processor does not do auto-refill in this case at all. It jumps right to the user or kernel exception vector because this is just considered an ordinary instruction-related exception. So, instead of an ITLB miss indication, PDebugStatus given an exception indication. The cause—InstTLBMissCause in this case—is reported on PDebugData[21:16] as described in Table 17–93.
- Miss on an auto-refill way, where the HW-assisted ITLB refill completes successfully: The ITLB miss is indicated first (in the W of the instruction that would have retired), followed some cycles later by the HW ITLB refill indication (in the W of the LDPTE instruction).

- Miss on an auto-refill way, where the HW-assisted ITLB refill *itself* encounters a DTLB miss: In this case, the ITLB miss is indicated first (in the W of the instruction that would have retired) but subsequently (in the W of the LDPTE instruction) an exception is indicated **with the original cause**—InstTLBMissCause—reported on PDebugData[21:16] as described in Table 17–93.
- Miss on an auto-refill way, where the HW-assisted ITLB refill encounters an unrelated exception e.g. DBREAK: In this case, the ITLB miss is indicated first (in the W of the instruction that would have retired) but subsequently (in the W of the LDPTE instruction) an exception is indicated **again with the original cause**—InstTLBMissCause—reported on PDebugData[21:16] as described in Table 17–93.

ITLB hits can be derived by subtracting ITLB misses from the number of retiring instructions. Even if the instruction goes through one of the no-translate/identity-mapped ways of the ITLB, that is still considered a hit. If there is an ITLB hit, and the instruction is killed for some other reason, that would not be considered a hit in this context.

This encoding is absent from configurations without full MMU.

17.3.10 DTLB Miss

PDebugStatus[5:0] = 0111_10 indicates a pipeline bubble caused by a DTLB miss. DTLB miss is analogous to ITLB miss, and the preceding description applies with D replacing I.

17.3.11 Stall Information

PDebugStatus[5:0] = 1000_00 indicates a global processor stall. This condition has priority for a couple of reasons.

- Why GlobalStall is happening is probably an important statistic desired by users. We want to indicate it regardless of other concurrently-occurring bubble conditions.
- The other bubble conditions are all tied to the W stage. GlobalStall obviously holds all stages and the sources may not all be W-stage signals.

The cause of the global stall is further classified in the PDebugData field as shown in Table 17–94.

While GlobalStall has the highest priority, it doesn't mean that other bubble conditions are lost. Consider a D-cache miss. In the W of the (killed) load or store, GlobalStall will not be asserted and the Traceport will indicate D-cache miss bubble (PDebugStatus[5:0] = 0011_00). In subsequent cycles, GlobalStall will by indicated (PDebugStatus[5:0] = 1000_00) with PDebugData[14] set.

Table 17–94. GlobalStall Cause Information

Field of PDebugData	Description
PDebugData[31:20]	Reserved
PDebugData[19]	Iterative divide stall. Execution of iterative divide instructions causes stall. Assertion of the stallout output of the IterativeIntegerDivider module causes this bit to be set.
PDebugData[18]	Iterative multiply stall. Execution of iterative multiply instructions causes stall. Assertion of the stallout output of the IterativeMultiplier module causes this bit to be set.
PDebugData[16]	Bank-conflict stall. It may be necessary to stall load/stores in the pipeline when there are competing accesses to the same bank. This includes inbound PIF requests (incl. RCW) to the given bank. Assertion of the BankConflictBusyLS*n* output of the LoadStoreUnit module causes this bit to be set.
PDebugData[15]	Bypass load stall. The LoadStore unit asserts stall when processing loads from bypass-attribute memory locations. (This includes the “load” associated with S32C1I.) It’s important to note that when this bit is asserted, PDebugData[14] is also always asserted because that indicates load/store processing stalls for all attributes. In other words, the stall information provided here is supplemental; to enable the differentiation of bypass-attribute vs. cached-attribute load/stores. Assertion of the UncachedStall_M output of the LSElement module causes this bit to be set.
PDebugData[14]	Load/store miss-processing stall. The LoadStore unit asserts stall when processing missing loads and stores, and in particular when looking up the miss-handling table (a logic structure inside Xtensa) in order to do this processing. As mentioned above, this includes load/stores to locations with any memory attribute—cached, bypass, etc. Assertion of the MHTLookupStall_M output of the LSElement module causes this bit to be set.
PDebugData[13]	FastL32R stall. The instruction fetch unit asserts this stall when creating a cycle for an L32R instruction to access the I-side local memories. Assertion of the Fastl32rStall output of the PCandIFetch module causes this bit to be set.
PDebugData[12]	Bypass I fetch stall. The PCUnit asserts this stall when fetching instructions from bypass-memory attribute. Assertion of the IFetchStallUncached output of the PCUnit module causes this bit to be set.
PDebugData[11]	Reserved
PDebugData[10]	RunStall. The pipeline stalls upon assertion of this pin to Xtensa. Note that the number of cycles of GlobalStall assertion is always one more than the number of cycles of RunStall assertion.
PDebugData[9]	TIE port stall. The pipeline may be stalled when executing instructions that access TIE ports such as queues or lookups. Assertion of the TIE_GlobalStall output of the TIE module causes this bit to be set. Refer to the description of the individual TIE port in the TIE RM for more detail.
PDebugData[8]	Instruction RAM inbound-PIF stall. The PC Unit may stall the pipeline when there is an inbound PIF request to one of the instruction RAMs. Assertion of the IRamDmaStall output of the PCUnit module causes this bit to be set.

Table 17–94. GlobalStall Cause Information (continued)

Field of PDebugData	Description
PDebugData[7]	Instruction RAM/ROM busy stall. In response to a fetch access, one of the local I memories may assert its busy, which may then in turn result in a pipeline stall. Assertion of the IMemBusyStall_P1 output of the PCUnit module causes this bit to be set.
PDebugData[6]	I-cache-miss stall. The PCUnit asserts this stall when servicing an instruction cache miss. Assertion of the IFetchStallCached output of the PCUnit module causes this bit to be set.
PDebugData[5]	Reserved.
PDebugData[4]	The LoadStore unit will stall the pipeline under various local memory access conflict situations. Examples are ▪ assertion of Busy input from the data local memory ▪ simultaneous access to D-cache in a 2-LoadStore config ▪ data local memory bank-conflict – this includes conflicts due to inbound PIF request (incl RCW) to one of the data RAMs (also on PDebugData[16]) ▪ instruction local memory access conflict when doing L32R therefrom (also on PDebugData[13]) These stall conditions are reflected by the LdStallBusy*n*_M output of the LoadStoreUnit module. Assertion of this signal causes this bit to be set. As indicated in the parentheses of the list items above, some of the components of this stall are reflected in other bits of PDebugData so that they may be individually counted. This bit incorporates all LS-related local memory access conflict stalls.
PDebugData[3]	D-cache-miss stall. In configurations without Early Restart, a lengthy data cache miss processing may result in stalls of the pipeline when the load/store instruction is being held in the R stage (i.e. DMissHold_R). Note that this stall is not included in the load/store miss-processing stall category of PDebugData[14]. Assertion of the DFetchStall output of the PCUnit module causes this bit to be set.
PDebugData[2]	Store buffer conflict stall. When a store in one LS unit has the same target as the store that is already in the store buffer of another LS unit, there may be a pipeline stall. Assertion of the St*n*matchQuandary_M output of the LSElement module causes this bit to be set.
PDebugData[1]	Store buffer full stall. When the store buffer of a LS unit is full, and a new store is present in the M stage, there may be a pipeline stall. Assertion of the St*n*owhereToGo_M output of the LSElement module causes this bit to be set.
PDebugData[0]	Reserved

Once again, PDebugData[3] applies when the instruction fetch unit does an R-stage stall for special, corner-case load/store miss processing. It does not include the GlobalStall while looking up the miss-handling table (a logic structure inside the Xtensa processor), which is a part of normal miss processing. The latter is indicated by PDebugData[14].

17.3.12 Hardware-Corrected Memory Error

PDebugStatus[5:0] = 1001_00 indicates that the hardware corrected a memory error on an instruction or data fetch and is applicable only if Memory Errors are configured.

17.3.13 WaitI Mode

PDebugStatus[5:0] = 1010_00 indicates that the processor has entered internal WAITI mode. The external PWaitMode signal will be asserted later than the WAITI encoding seen on Traceport. How many cycles later is not fixed, and depends on transactions in flight on Xtensa processor interfaces such as TIE ports or PIF.

17.3.14 Other Bubbles

PDebugStatus[5:0] = 1111_00 indicates all other pipeline bubbles. These bubbles are often associated with previous bubble causes. For example, an instruction-cache miss will cause PDebugStatus[5:0] to be set to 0010_00 during the cycle corresponding to the W stage of the instruction that causes the cache miss. This will be followed by a number of cycles where PDebugStatus[5:0] = 1111_00, representing the additional pipeline bubbles caused by the original instruction-cache miss.

17.3.15 Special Use of PDebugPC

In certain bubble cases, Xtensa uses PDebugPC for purposes not immediately intuitive. In particular, TRAX requires a valid PC even when there is no valid instruction of exception signaled on the Traceport. It needs this to correctly construct sync messages for instance. To accommodate this the Traceport speculates the PC as follows

- in the cycle after a branch or exceptOrInt_X, if there is no valid instruction, PDebugPC shows the target PC because that is likely the next retiring PC
- in the cycle after a valid instruction, if there is no valid instruction - PDebugPC shows the sequential PC

17.4 Instruction Types

When PDebugInst[7:0] is not 8’b0, a valid instruction is retiring in the W stage. PDebug-Status[5:0] gives the type of instruction as shown Table 17–95. PDebugStatus[7:6] is reserved.

Table 17–95. Types of Instructions

PDebusStatus [5:0]	Description	PDebugData	PDbg PC
0000_00	JX	Target virtual address	**0000_00**
0001_00	CALLX	Target virtual address	0001_00
0010_00	All call returns	Target virtual address	0010_00
0011_00	All exception returns	Target virtual address	0011_00
0100_00	Branch taken or loop not taken	Target virtual address	0100_00
0101_00	J	Target virtual address	0101_00
0110_00	CALL	Target virtual address	0110_00
0111_00	Branch not taken	*undef*	0111_00
1000_00	Loop instruction (taken)	LBEGIN virtual address	1000_00
1001_00	S32C1I	Conditional store data	1001_00
1010_00	WSR/XSR to LBEGIN	SR Write data value	1010_00
1011_00	WSR to MMID	SR Write data value	1011_00
1100_00	RSR or WSR (except MMID and LBEGIN) or XSR (except LBEGIN)	SR number and inst type (see Table 17–96)	1100_00
1101_00	RER or WER	ER address and inst type (see Table 17–97)	1101_00
1110_xx	Reserved	*undef*	1110_xx
1111_00	Default (none of the above)	*undef*	1111_00

17.4.1 Instruction Virtual Address

The PC interface, PDebugPC[31:0], indicates the value of the program counter when an instruction completes in the Xtensa processor pipeline; as indicated by PDebugInst not equal to 8'b0. When PDebugInst[7:0] is 8'b0, this interface is undefined, except for exceptions/interrupts when PDebugStatus[5:0]==010000.

17.4.2 Control Transfer Instructions

PDebugStatus[5:0] = 0100_00 denotes a taken branch, and PDebugStatus[5:0] = 0111_00 denotes a branch not taken.

PDebugStatus[5:0] = 0100_00 also indicates a loop instruction where the loop is not taken. Loop instructions LOOPGTZ and LOOPNEZ behave like a branch if the loop condition is not satisfied. PDebugStatus[5:0] = 1000_00 indicates a taken loop and PDebugStatus[5:0] = 1010_00 indicates writes to the LBEGIN Special Register.

17.4.3 S32C1I

1001_00 is output on PDebugStatus[5:0] for a `S32C1I` instruction. The conditional store data is available on PDebugData even if the store is not actually performed i.e. the compare fails.

The address and load data (before the conditional store) are available on the LS signals as shown in Table 17–101.

17.4.4 Special Registers

1010_00 is output on PDebugStatus[5:0] for a `WSR.LBEG` instruction, 1011_00 is output on PDebugStatus[5:0] for a `WSR.MMID` instruction, and 1100_00 is output for writes to all other special registers.

MMID, (memory map ID) is a 32-bit wide write-only special register (SR number 89) that is configured only when the Traceport option is selected. The MMID register contents are reflected on the Traceport to help in decoding the trace output by defining which memory map is in force.

1010_00 is output on PDebugStatus[5:0] for an `XSR.LBEG` instruction, and 1100_00 is output for XSR instructions directed at all other special registers.

When PDebugStatus[5:0] = 1100_00, PDebugData indicates the SR number and instruction type as shown in Table 17–96 below. Also, the RSR/WSR/XSR read/write data are available on the LS signals as shown in Table 17–101.

Table 17–96. SR Number and Instruction Type

Field of PDebugData	Description
PDebugData[31:11]	Reserved
PDebugData[10]	XSR Instruction

Table 17–96. SR Number and Instruction Type (continued)

Field of PDebugData	Description
PDebugData[9]	WSR Instruction
PDebugData[8]	RSR Instruction
PDebugData[7:0]	Special Register Number

17.4.5 External Registers

When PDebugStatus[5:0] = 1101_00, PDebugData indicates the ER address and instruction type as shown in Table 17–97. External registers are 32 bits wide, and addressed using a 30-bit address, (i.e. the least significant two bits of the AR are don't-care) therefore the complete address is potentially available. For RE.0 (and possibly future releases) however, only the bottom 12 bits are put out i.e. as with APB, 16K page size.

Table 17–97. ER Address and Instruction Type

Field of PDebugData	Description
PDebugData[13:2]	ER address
PDebugData[1]	WER address
PDebugData[0]	RER Instruction

17.4.6 Other Instructions

Instructions that are not branches, loops, loads, or stores are classified as default instructions where PDebugStatus[5:0] = 1111_00. If a user-defined TIE instruction is a branch, load, or store instruction, then the corresponding PDebugStatus value is output. Otherwise, the instruction is classified as a default instruction and PDebugStatus[5:0] is set to 1111_00.

17.5 Instruction Status

PDebugInst gives the instruction or fetch status of each retiring instruction. A pipeline bubble is indicated by PDebugInst[7:0]==8'b0. The other fields of PDebugInst are valid only when PDebugInst[7:0] is non-zero.

Table 17–98. Instruction Status

Field of PDebugInst	Description	Encoding
PDebugInst [7:0]	Instruction size	number of bytes, e.g. 8'b0000_0110 if 48-bit wide instruction
PDebugInst [11:9]	Reserved	
PDebugInst[14:12]	Instruction source (field present only when config has >1 local inst mem)	000: reserved or not implemented 001: PIF or OCD 010: I-cache 011: InstRAM0 100: InstRAM1 101: InstROM0
PDebugInst [15]	Reserved	
PDebugInst [19:17]	Reserved	
PDebugInst[23:20]	Loopback status [21] last inst of loop [20] loopback will occur	xx0x: not the last inst of either loop xx10: last inst of loop, no loopback xx11: last inst of loop, w loopback
PDebugInst [27:24]	CINTLEVEL	as defined in ISA

17.5.1 Instruction Size

The size of the instruction in the W stage is given by PDebugInst[7:0]. The size is simply expressed in bytes, e.g. 8'b0000_0011 if the instruction is 24-bit wide. PDebugInst[7:0] is 8'b0 if in a given cycle the W stage has no completing instruction.

17.5.2 Instruction Source

PDebugInst[14:12] gives the source of the instruction. This field is present only in configurations with more than one local instruction memory e.g. two instruction RAMs, InstRAM0 and InstROM, InstRAM0 and I-cache, etc. For the purposes here, an I-cache is considered a local memory. In configurations with only one local instruction memory, the field has value 4'b0.

The information in PDebugInst[14:12] can for instance be used to count fetches from IRAM0—as opposed to other sources. It's possible that an instruction comes from two sources—for example, when an instruction straddles an IRAM boundary. In such cases PDebugInst[14:12] refers to the trailing end of the instruction.

PDebugInst[14:12]==010 means that the instruction was supposed to hit in the I-cache, and it did.

I-cache miss information is available from the bubble information of Table 17–90 corresponding to the killed W. Upon the replay—and assuming that the instruction actually hits this time and retires properly—Hit will be true in the W-stage.

If both Miss and Hit are false, look elsewhere for information regarding the killed or retiring instruction.

To count the true number of I-cache hits, a user would have to subtract the misses from the hits. This can be inaccurate in that the replaying instruction could be killed—for example, because of an interrupt.

PDebugInst[14:12]==001 means that the instruction came from an uncached PIF fetch. Uncached PIF transactions are countable through the bubble information provided on PDebugStatus as described in section 3.4. However, the two numbers would not match because the former corresponds to retiring instructions whereas the latter corresponds to uncached PIF transactions.

PDebugInst[14:12] also signals that the instruction is from OCD, which could be used by agents such as TRAX.

17.5.3 Loopback Status

For every valid instruction i.e. PDebugInst not 0, PDebugInst[22:20] provides loopback information.

In the case where the last instruction of a loop is a branch, LCOUNT is undefined and this branch must be used to exit the loop. Because LCOUNT is undefined, the Traceport hardware will indicate a taken branch on PDebugStatus and PDebugStatus[7:6] will be set to 3'b000.

17.6 Load/Store Instructions

For each load-store unit configured, selecting the DataTrace option gives three 32-bit signals, PDebugLS*n*Stat, PDebugLS*n*Addr and PDebugLS*n*Data.

17.6.1 Load/Store Status

As shown in Table 17–99, PDebugLS*n*Stat gives detailed information about the ld/st operation in the W stage, regardless of hit or miss. More explicitly, the fields of PDebugLS*n*Stat are valid:

- when there is a retiring load or store in the W stage
- when there is a killed load or store in the W stage, if there is a LS reason for the kill, such as D-cache miss
- when there is a HW-assisted TLB refill instruction in the W-stage

In these cases, PDebugLS*n*Stat[3:0] will have a non-zero value. This characteristic is different from PDebugInst, whose fields (other than the instruction size PDebugInst[7:0]) are valid only when there is a valid instruction retiring in the W stage.

The fields of PDebugLS*n*Stat are not valid

- when there is a non-ld/st instruction in the W stage, regardless retiring or killed
- when there is a killed load or store in the W stage, if the reason for the kill was not related to LS
- regardless of valid LS operations/functions in other stages of the pipeline e.g. bank-conflict detection, tag lookup, etc.
- when GlobalStall is being asserted, even if the original cause of the GlobalStall was LS-related e.g. D-cache miss

In these cases, PDebugLS*n*Stat[3:0] will be zero.

Table 17–99. Load/Store Information

Field of PDebugData	Description	Encoding
PDebugLS*n*Stat[3:0]	Type of instruction in LS*n*	0000: neither 0001: hw itlb refill 0010: hw dtlb refill 0011: reserved 0100: reserved 0101: load 0110: store 0111: reserved 1000: l32r 1001: reserved 1010: s32ci1 1011: reserved 1100: cache test inst 1101: reserved 1110: rsr/wsr/xsr 1111: rer/wer
PDebugLS*n*Stat[7:4]	Operand size	0000: 8-bit data 0001: 16-bit data 0010: 32-bit data 0011: 64-bit data 0100: 128-bit data 0101: 256-bit data 0110: 512-bit data 0111 and above: reserved
PDebugLS*n*Stat[8]	Data TLB miss	0: not a miss 1: miss
PDebugLS*n*Stat[9]	D-cache miss	0: not a miss 1: miss
PDebugLS*n*Stat[10]	D-cache hit	0: not a hit 1: hit
PDebugLS*n*Stat[11]	Reserved	
PDebugLS*n*Stat[12]	Uncached	0: not uncached 1: uncached
PDebugLS*n*Stat[13]	Writeback	0: not writeback 1: writeback
PDebugLS*n*Stat[15:14]	Reserved	

Table 17–99. Load/Store Information (continued)

Field of PDebugData	Description	Encoding
PDebugLS*n*Stat[16]	Coherency	0: non-coherent 1: coherent
PDebugLS*n*Stat[18:17]	Coherent state degenerates to [17] dirty [18] valid in non-Coherency configs	00: neither shared nor exclusive nor modified 01: shared 10: exclusive 11: modified
PDebugLS*n*Stat[23:20]	Local target	0000: not to local memory 001x: InstRAM (0/1) 010x: InstROM (0/1) 101x: DataRAM (0/1) 111x: DataROM (0/1) others: reserved
PDebugLS*n*Stat[31:28]	Reserved	

17.6.1.1 Type of LS instruction

PDebugLS*n*Stat[3:0] shows the type of LS instruction. The encoding shows that the user gets both retired ld/st W-stage information as well as killed ld/st W-stage information.

17.6.1.2 Size

For TIE load or store instructions that use arbitrary byte enables, PDebugLS*n*Stat[7:4] will not indicate the actual number of bytes accessed. For example, if the instruction semantic uses the 4-byte load interface and sets "1100" for the Byte Disable interface, PDebugLS*n*Stat[7:4] will indicate "0010" and not "0001". There is one exception to this rule—if the load or store is entirely killed because the Byte Disable interface is set to "0000," PDebugLS*n*Stat[3:0] will indicate "0000".

17.6.1.3 TLB Miss

PDebugLS*n*Stat[8] reports DTLB misses. DTLB hit information is derivable from the number of successfully retiring ld/st instructions minus DTLB miss information. No translation — i.e. going through one of the DTLB's no-translation/identity-mapped ways — is still regarded as a hit if the ld/st retired.

This bit is absent/undefined in configurations without full MMU.

17.6.1.4 Miss and Hit

PDebugLS*n*Stat[9] indicates D-cache miss and PDebugLS*n*Stat[10] indicates D-cache hit. In this context, Miss is: load/store was supposed to hit in the cache, but it did not. Note that this is the same information as reported by (for LS0 for instance) PDebugData[4] (Table 17–92) when a D-cache miss bubble is indicated by PDebugStatus[5:0] = 0011_00 (Table 17–90). Similarly, Hit is defined: load/store was supposed to hit in the cache, and it did. The Miss bit will be true in the W stage of the load or store—again that was supposed to hit in the cache—that was killed. Upon the replay—and assuming that the load/store actually hits this time and retires properly—the Hit bit will be true in the W-stage.

If both Miss and Hit are false, look elsewhere for information regarding the killed or retiring load/store.

Miss and Hit cannot both be true in the same cycle.

To count the true number of cache hits, a user must subtract the misses from the hits. This can be inaccurate in that the replaying load/store could be killed—for example, because of an interrupt—but this effect does not grossly distort the miss/hit rate calculation.

17.6.1.5 Uncached

PDebugLS*n*Stat[12] indicates an uncached load or store. Uncached is a load from the PIF because the memory attributes said “Bypass”, or the memory attributes said “No Allocate” and the load missed in the cache. This is only indicated the first time around, when the load is killed. When the load is successfully retired, there is no indication as to the target of the load. Counting upon the killed W gives a more accurate count of uncached loads because they are interruptible. Inaccuracies would only be caused by bus error.

Not Uncached does not mean cached.

For stores, Uncached is used to distinguish local RAM stores from Bypass ones. Also as Table 17–100 shows, it’s used to distinguish No Allocate stores from Allocate stores.

17.6.1.6 Writeback

PDebugLS*n*Stat[13] indicates a Writeback load or store. Writeback is derived from the WriteThru memory attribute. It’s meaningful only for stores. Combined with the D-cache hit information, it can be used—where the user expects a high D-cache write hit rate—to see whether the core is generating unduly high PIF store traffic. Similarly, combined with the D-cache miss information, it can be used to see whether stores are causing unnecessary refills—i.e. store would have been better off to WriteThru region.

Stating the obvious, writeback miss is only indicated upon the killed W of the store. When the store comes round again upon the replay, it would (under normal circumstances) be a writeback hit.

Table 17–100. Semantics of Miss, Hit, Uncached, and Writeback

Miss	Hit	Unc	WB	Store Semantics
0	0	0	0	store to RAM/ROM/XLMI
0	0	1	0	Bypass store, to PIF only
0	1	0	0	WT Alloc not currently supported
0	1	0	1	WB Alloc store hit, to D-cache only
0	1	1	0	WT NA store hit, to both D-cache and PIF
0	1	1	1	WB NA store hit, to D-cache only
1	0	0	0	WT Alloc not currently supported
1	0	0	1	WB Alloc miss, initiate refill
1	0	1	0	WT NA miss, to PIF only
1	0	1	1	WB NA miss, to PIF only

17.6.1.7 Local Target

PDebugLS*n*Stat[23:20] signals that the ld/st target is a local memory. This information can for instance be used to count load/stores to DataRAM0—as opposed to other targets.

Note that local memory accesses are derivable from the address available on PDebugLS*n*Addr, but we have this information anyway for redundancy and ease of counting (i.e. no need for comparator) by external hardware.

17.6.2 Load/Store Address and Data

As shown in Table 17–101, PDebugLS*n*Addr and PDebugLS*n*Data provide the address and data of the operation in the load/store unit. It might be necessary to use a combination of PDebugData and PDebugLS*n*Stat to distinguish the classes of operations listed in the table. Cache Access instructions includes DPFL. The cache address value that appears on PDebugLS*n*Addr for these instructions is the data value of the AR register used by that cache-test instruction.

PDebugLS*n*Data is always fixed at 32 bits, regardless of the true LSwidth. For loads and stores wider than 32 bits, only the least significant 32 bits are available. “Least significant” here refers to the post-rotated MemDataIn or the pre-rotated MemDataOut. In other words, when the data goes to, or comes from, an AR, the 32 bits of the AR will be available on PDebugLS*n*Data regardless of LSwidth, or memory endianness.

Table 17–101. Load/Store Address and Data

PDebugData and PDebugLSnStat	PDebug LS*n*Addr	PDebug LS*n*Data
Miss	Data virtual address	undef
TLB Refill (both I and D)	Load virtual address	TLB write data
Load	Data virtual address	Load data (or 32 least significant bits thereof)
Store	Data virtual address	Store data (or 32 least significant bits thereof)
S32C1I	S32C1I virtual address	Load data (indep of compare)
Cache Access Insts	Cache address	Cache ld or st data (depends upon op)
RSR / XSR / WSR	XSR write data value WSR write data value	RSR read data value XSR read data value
RER / WER	WER write data value	RER read data value
None of the above	*undef*	*undef*

17.7 Pipeline-Independent Traceport Signals

The Traceport signals events other than those associated with instructions in the Xtensa pipeline. Specifically, PIF-based transaction information is signaled on PDebugOutPIF and PDebugInbPIF; prefetch-related trace data is signaled on PDebugPrefetchLookup and PDebugPrefetchL1Fill; and iDMA trace information is signaled on PDebugiDMA. While the instruction-related Traceport signals are tied to the W stage of the pipeline, these pipeline-independent signals are meaningful only when specific function-related events occur. For example, PDebugOutPIF and PDebugInbPIF are tied to either the request or response of the PIF transaction.

With the pipeline-independent Traceport signals, the idea is not to duplicate what is already available on external interfaces, but to provide more Xtensa-internal information. For example, with PIF-related trace data the request to response-ready time might be a useful performance metric, but it's easily observable on the PIF, so it is not part of PDebugOutPIF.

17.7.1 Outbound PIF Transactions

PDebugOutPIF provides information on outbound PIF requests as shown in Table 17–102. The information is provided at the start of the request—more specifically when both POReqValid and PIReqReady are asserted.

Again, some of PDebugOutPIF is indeed information available through the POReqCntl, POReqID, etc. bits of the PIF request. However, the organization is a little different, to allow for future expansion and extraction of information not directly available.

Table 17–102. Outbound PIF trace

Field	Description	Encoding
PDebugOutPIF[7:6]	Reserved	
PDebugOutPIF[5]	Prefetch	0: non-prefetch 1: prefetch
PDebugOutPIF[4]	Castout	0: not a castout 1: castout
PDebugOutPIF[1:0]	Request type	00: no outbound PIF 01: request for read 10: request for write 11: request for RCW

PDebugOutPIF[5] says whether a prefetch is being requested.

PDebugOutPIF[4] says whether a block write due to a castout is being requested on the PIF by the Xtensa processor. Although we use the term “castout” here, this is actually an indication only of a block write. If *loadstorewidth* is greater than *pifwidth*, a single store is going to be converted to a block—i.e., even though it's a block, it's not necessarily a castout. Conversely, if *pifwidth* is the same as *linesize*, a castout will be seen as a single write.

Clearly, not all encodings of PDebugOutPIF are valid.

PDebugOutPIF[3:2] is reserved for future use.

17.7.2 Inbound PIF Transactions

PDebugInbPIF provides information on inbound PIF requests as shown in Table 17–103. The information is generated:

- in the cycle a read single is being issued to the target and a previous read or write is not being retried
- in the cycle the last word of a block read is being issued to the target and a previous read or write is not being retried
- in the cycle a write single is being issued to the target and a previous read or write is not being retried
- in the cycle the last word of a block write is being issued to the target and a previous read or write is not being retried
- in the cycle that the RCW to the target successfully completes

Stating the obvious, for a block, the request is indicated only once—and not once per word for instance.

Table 17–103. Inbound PIF trace

Field	Description	Encoding
PDebugInbPIF[7:6]	Target	11: InstRAM1 10: InstRAM0 01: DataRAM1 00: DataRAM0
PDebugInbPIF[5:3]	Reserved	
PDebugInbPIF[2]	Block vs. single	0: single 1: block
PDebugInbPIF[1:0]	Request type	00: no inbound PIF 01: request for read 10: request for write 11: request for RCW

The target (PDebugInbPIF[7:6]) and block (PDebugInbPIF[2]) fields are valid only when the request field (PDebugInbPIF[1:0]) is non-zero.

17.7.3 Prefetch

PDebugPrefetchLookup provides information about the number of lookups of the prefetch buffers generated by the pipeline in response to a cache miss—or due to a software-directed prefetch instruction. The bus also provides more detail about the type of lookup as shown in Table 17–104 below.

Table 17–104. Prefetch lookup

Field	Description	Encoding
PDebugPrefetchLookup[7:4]	Reserved	
PDebugPrefetchLookup[3]	Hit but waiting for response. If "hit" is true, whether the prefetch response has arrived or not.	0: waiting for prefetch response 1: response has fully arrived
PDebugPrefetchLookup[2]	Hit in prefetch buffers. Hit indicates that prefetch processing has already been initiated. Miss indicates that a brand new prefetch request will be initiated (on the PIF) and maintained by the prefetching engine.	1: hit 0: miss
PDebugPrefetchLookup[1]	Instruction vs. Data	0: instruction 1: data
PDebugPrefetchLookup[0]	Lookup. Regardless of any stall conditions, asserted only once for every valid lookup.	1: valid lookup 0: no lookup

PDebugPrefetchL1Fill provides information about fills to the L1 cache from the prefetch buffers as shown in Table 17–105 below. Fills to the L1 obviously reduce the number of lookups to begin with. Note that not all prefetches are destined to L1—e.g. instruction prefetch or control bit DL1 is clear.

The information provided on PDebugPrefetchL1Fill is at the time of successful completion of the L1 fill. So, fills that were canceled mid-stream are not signaled on the Traceport.

Table 17–105. Fills to L1 cache

Field	Description	Encoding
PDebugPrefetchL1Fill[3:1]	Reserved	
PDebugPrefetchL1Fill[0]	Fill to L1. Regardless of any stall conditions, asserted only once for every valid fill completion.	1: valid fill 0: no fill

17.7.4 Integrated DMA

PDebugiDMA provides information on the status of iDMA transactions as shown in Table 17–106.

Table 17–106. Integrated DMA trace

Field	Description	Encoding
PDebugiDMA[7:3]	Reserved	
PDebugiDMA[2:0]	Run Mode	000: idle 001: standby 010: busy 011: done 100: halt 101: error Other encodings are reserved

PDebugiDMA contains only one field. PDebugiDMA[2:0] directly traces the Run Mode of the iDMA control FSM. This information can be used by external logic to know for example when the FSM has no more descriptors to work on.

As with other Traceport signals, the values seen on the PDebugiDMA are delayed due to flopping inside Xtensa. In the PDebugiDMA case, it is one cycle.

18. Xtensa Trace Display Tool

The software tool `xt-trace` converts raw signal information captured at the Xtensa Traceport into a stream of assembly instructions. It also offers varying levels of processor-state information.

This chapter discusses all command-line arguments of `xt-trace` in detail, offers examples, and identifies a few caveats.

18.1 Introduction

Invoking `xt-trace` with no arguments generates the following statement and quits.

```
Warning: option --trace is required
Usage: xt-trace --trace <file> --executable <file>
                [--output <file>]
                [--fields <field>,<field>...]
                [--with-bubbles]
                [--count-bubbles]
                [--symbolic]
                [--no-symbols]
                [--reverse]
                [--version {TP4}]
                [--echo-input]
                [--help]
Note: options may be abbreviated to a unique prefix.
```

18.2 Command-Line Arguments

This section explains the command-line arguments listed in the usage statement in Section 18.1.

18.2.1 Executable File (--executable filename)

The `--executable` argument specifies the executable file that is running. It is typically the file that the linker generates as the last step of the build process, and is in ELF format. `xt-trace` uses this executable file to disassemble the opcodes of the instruction virtual addresses that come from the Xtensa Traceport.

18.2.2 Data File (--trace filename)

A hardware simulator or logic analyzer can capture the raw signal information from the Xtensa Traceport and save the captured data to a text file. `xt-trace` converts the data file to a format more suitable for human consumption.

`xt-trace` expects the input file to contain several columns of data in hexadecimal format separated by spaces. This data provides the state of the Traceport, one line per processor cycle. The first line may optionally contain column headers which specify the meaning of each column of data. The header line may optionally be followed by a line of non-hex data as a separator.

If the first line of the input file is data, the meaning of the data must be specified via the `--fields` option for (see Section 18.2.4).

The following table lists the accepted forms for the various field names:

Table 18-107. Traceport Data Field Names

Field	Aliases
PDebugInst	PInst, Inst
PDebugStatus	PStatus, Status
PDebugData	PData, Data
PDebugPC	PC
PDebugLS0Stat	LS0Stat
PDebugLS0Addr	LS0Addr
PDebugLS0Data	LS0Data
PDebugLS1Stat	LS1Stat
PDebugLS1Addr	LS1Addr
PDebugLS1Data	LS1Data

Following are the first few lines of a TP4-style input file:

```
    PInst PStatus PDebugData       PC  LS0Stat  LS0Addr  LS0Data
--------- ------- ---------- -------- -------- -------- --------
        0      00          0        0        0        0        0
        0      00          0        0        0        0        0
        0      00          0        0        0        0        0
        0      00          0        0        0        0        0
        0      3C   40000000        0        0        0        0
 0f000000      3C   40000000        0        0        0        0

    PInst PStatus PDebugData       PC  LS0Stat  LS0Addr  LS0Data
--------- ------- ---------- -------- -------- -------- --------
```

```
0f000000      3C   40000000        0        0        0        0
0f000000      20   00001000        0        0        0        0
0f000000      20   00001000        0        0        0        0
  ...         ..     ...       ...      ...      ...        .
0f001003      14   4000000c 40000000        0        0        0
  ...         ..     ...       ...      ...      ...        .
0f001000      0c   00000011 4000000c 00001228 40000008        0
0f001000      08   40000010 4000000c 00000020 40000008        0
0f001000      3c   4000000c 4000000c 00000020 40000008        0
0f001000      3c   4000000c 4000000c 00000020 40000008        0
0f001000      20   00001000 4000000c 00000020 40000008        0
0f001000      20   00001000 4000000c 00000020 40000008        0
```

Note: Only hexadecimal digits (and 'X') are recognized as valid in the input stream after the optional header lines. X's are treated like zeros.

`xt-trace` ignores the following:

- Extraneous white space
- Any line contents following the hexadecimal value for the last column

18.2.3 (Output File (--output filename)

This argument indicates that `xt-trace` should place its output in *filename* instead of sending it to `stdout`.

The following is an example of the output using the default options with an Xtensa LX5 configuration.

```
XTENSA_CORE = Software, Traceport Version = TP4
------- -------------------------
Cycle   Instruction or other info
------- -------------------------
        At <_ResetVector>
      7 20000020:       j       0x2000003c <_ResetHandler>
        Jump to 0x2000003c <_ResetHandler>
        At <_ResetHandler>
     10 2000003c:       movi    a0, 0
     11 2000003f:       wsr.intenable   a0
        Special register 228 <- 0x0
     12 20000042:       wsr.ccount      a0
        Special register 234 <- 0x0
     13 20000045:       wsr.dbreakc0    a0
        Special register 160 <- 0x0
     14 20000048:       wsr.dbreakc1    a0
        Special register 161 <- 0x0
```

```
15 2000004b:       rsr.icountlevel a2
   Special register 237 -> 0x0
17 2000004e:       bltui   a2, 12, 0x20000054 <_ResetHandler+24>
   Branch to 0x20000054 <_ResetHandler+24>
20 20000054:       isync
27 20000057:       movi.n  a2, 1
28 20000059:       wsr.ps  a2
   Special register 230 <- 0x1
29 2000005c:       rsync
32 2000005f:       movi    a2, 128
33 20000062:       movi    a3, 0
34 20000065:       loop    a2, 0x20000077 <_ResetHandler+59>
   Beginning Zero Overhead Loop at 0x20000068
35 20000068:       iii     a3, 0
42 2000006b:       iii     a3, 16
49 2000006e:       iii     a3, 32
55 20000071:       iii     a3, 48
61 20000074:       addi    a3, a3, 64
    (Looping back)
63 20000068:       iii     a3, 0
```

The `Cycle` field is a sequential counter that begins with one and increments by one every cycle. Gaps in the sequence of cycle values are due to bubble cycles as explained below (Section 18.2.5).

The `Bub` field (not shown above; see `--count-bubbles` in Section 18.2.6) displays the number of bubble cycles that occurred immediately prior to this cycle, or blanks if none.

The `Instruction or other info` field contains an instruction's virtual address and opcode mnemonic plus operands. It may also contain information concerning calls, branches, loads, stores, or exceptions.

18.2.4 Field Names (--fields)

If the first line of the input file does not contain the names of the fields, you may specify the field names using the `--fields` option. The argument is a list of field names, separated by commas (or spaces, if quoted). For example, the data given in Section 18.2.2 on page 326 could be described as follows:

```
--fields Status,Data,PC,LS0Stat,LS0Addr,LS0Data
```

18.2.5 Display Pipeline Bubbles (--with-bubbles)

The `--with-bubbles` argument enables full reporting of pipeline-bubble cycles. By default, `xt-trace` filters pipeline-bubble cycles from the output. Note that this filtering causes gaps in the cycle count (which is appropriate, since the bubble cycles are filtered out).

18.2.6 Count Bubbles (--count-bubbles)

The `--count-bubbles` option adds a column to the output labeled `Bub` that shows the number of pipeline-bubble cycles that immediately preceded each instruction. For example:

```
XTENSA_CORE = Software, Traceport Version = TP4
------- --- -------------------------
Cycle   Bub Instruction or other info
------- --- -------------------------
            At <_ResetVector>
      7   6 20000020:   j       0x2000003c <_ResetHandler>
            Jump to 0x2000003c <_ResetHandler>
            At <_ResetHandler>
     10   2 2000003c:   movi    a0, 0
     11     2000003f:   wsr.intenable   a0
            Special register 228 <- 0x0
```

18.2.7 Additional Debug Information (--symbolic)

The `--symbolic` option enables symbolic address decoding on every disassembly line. By default, symbolic address decoding is provided only as needed, for optimal readability. For example, the symbol associated with the current instruction's virtual address is displayed after taken branches and calls.

The first part of the example given in Section 18.2.3 on page 327 would look like this if the `--symbolic` option were present:

```
XTENSA_CORE = Software, Traceport Version = TP4
------- -------------------------
Cycle   Instruction or other info
------- -------------------------
        At <_ResetVector>
      7 20000020 <_ResetVector>:        j 0x2000003c <_ResetHandler>
        Jump to 0x2000003c <_ResetHandler>
        At <_ResetHandler>
     10 2000003c <_ResetHandler>:       movi    a0, 0
     11 2000003f <_ResetHandler+3>:     wsr.intenable   a0
        Special register 228 <- 0x0
```

18.2.8 Suppress Symbolic Lookup (--no-symbols)

The `--no-symbols` option completely disables symbolic address decoding. This produces output like the following:

```
XTENSA_CORE = Software, Traceport Version = TP4
------- -------------------------
Cycle   Instruction or other info
------- -------------------------
      7 20000020:       j       0x2000003c
        Jump to 0x2000003c
     10 2000003c:       movi    a0, 0
     11 2000003f:       wsr.intenable   a0
        Special register 228 <- 0x0
```

18.2.9 Traceport Version (--version)

The `--version` option forces `xt-trace` to assume a particular Traceport version. Otherwise, `xt-trace` uses the Traceport version corresponding to the targeted processor configuration. This option is normally unnecessary and omitted.

18.2.10 Echo Input (--echo-input)

The `--echo-input` option causes `xt-trace` to echo each line of input, prefixed by the characters `INPUT:`, immediately before the corresponding output for that line.

18.3 Caveats of Using `xt-trace`

`xt-trace` emits appropriate messages in the output stream when it encounters data that has no meaning. Frequent occurrences of these messages are symptomatic of corrupted trace capture (as in loose or misplaced logic-analyzer probes) or misaligned data (for instance, the endianness is backward).

19. Xtensa Debug Monitor (XMON) for Xtensa LX Processors

Note: The Xtensa Debug Monitor (XMON) is available for Xtensa LX processors only.

XMON is a GDB debug monitor stub for Xtensa architecture processors. It is delivered as a library, which can be included in the user application. Essentially, XMON implements the Debug Exception Vector to handle debug exceptions.

XMON handles common GDB protocol requests such as reading and writing memory and registers, single-stepping, running, stopping (with Ctrl-C if a device interrupt is available), and breakpoints.

XMON talks to GDB over a customer-defined link, such as UART or TCP/IP. The main application must provide the driver for the communication link (usually in polling mode), including setting up an interrupt handler to enable GDB to stop a running core (e.g., with Ctrl-C) and obtain control over the application.

XMON readily works on bare-metal systems without an OS or with just XTOS, or with most RTOS. XMON is independent of the OS, so unlike debug agents integrated in the OS such as Linux gdbserver, it does not report OS features such as threads.

19.1 XMON Overview

The XMON library, when linked with an application, implements the Debug Exception vector and handler. XMON is thus entered on debug exceptions and interrupts. On processors configured with On-Chip Debug (OCD), such debug exceptions and interrupts are handled differently if OCD is enabled. In cases where OCD is enabled, the processor enters the debug *Stopped* state so that it can be controlled by an external debugger such as XOCD (see Section 7.1 “Debugging Methods Terminology”). If OCD is disabled, debug events are vectored to XMON, just as if OCD was not configured.

XMON allows debugging of an application without the need for XOCD and JTAG, or a standalone debug monitor. However, it comes at a cost, increasing the application size (on the order of 20KB as of this writing).

XMON readily provides the ability to debug an application by linking in its library, and providing the communication link driver functions described in Section 19.3.

The application can be a standalone debug monitor, stored perhaps in ROM or flash, that GDB can connect to in order to download and debug other applications. If all the debug monitor does is support XMON (no console, etc.), it can be as simple as a tiny `main()` program looping on a break instruction. However, using XMON to download and run a separate application (which must also link in XMON if debugging is to continue

there), or to reload the currently running application, entails careful management of the memory map, and possibly custom relocation mechanisms, to avoid the running and downloaded applications overlapping each other in memory during download. Such a scenario is outside the scope of this document or of XMON. Briefly, some example methods of avoiding this overlap include:

- Using separate non-overlapping memory maps or LSPs (see the *Xtensa Linker Support Packages (LSPs) Reference Manual*) for each application, possibly with non-overlapping vectors as well; and/or,
- A custom linker script and startup code that relocates the application when it starts.

As previously stated, XMON does need supporting routines from the application to set up the link used to communicate with GDB. This includes:

- A driver for the I/O device used (such as UART or TCP/IP). This is typically a polling mode driver, so that XMON can disable interrupts to stop the whole processor.
- A set of user-defined functions (see Section 19.3) used by XMON to interface to this driver, that is, to send and receive characters to and from GDB.
- An interrupt handler for the I/O device used, triggered upon receiving any character, so that GDB can stop the target, for example when you key Ctrl-C. Fortunately, this handler can be as simple as a single `break` instruction, whose execution transfers control to the Debug Exception Vector, that is, to XMON.

A very simple example of such a driver for the UART on Xilinx ML605 and KC705 emulation boards can be found in Xtensa Tools directory:

```
<xtensa_tools_root>/xtensa-elf/src/xmon/xmon-main-serial.c
```

In contrast to the debugging using OCD, XMON has several benefits and drawbacks.

Not being limited to JTAG (or debug APB) can allow for easier in-the-field debugging, as XMON may communicate with GDB using the I/O the system already contains. Also, XMON can provide easier security as access to the communication link can be controlled by application software, whereas JTAG/APB allows full external control of the core if no special hardware prevents it. Finally, when the XMON is active (GDB has control), interrupts of higher priority than DEBUGLEVEL can still be taken, which is important in systems with real-time events that must be serviced regardless of debugging activities in progress.

Conversely, debug monitors such as XMON are intrinsically more intrusive than on-chip debug (OCD): they take up memory space, and alter the state of the system (e.g., caches and memory) while processing debug events. XMON relies on the application successfully running far enough to initialize the communications link and XMON itself. In its current incarnation, XMON implements single-core debug.Although you can link XMON

in application images (each running on separate cores and each with their own communications link to a GDB instance), note that XMON does not provide synchronization of cores or multiplexing of the communications link.

19.2 XMON-provided Routines

The following routines are provided by XMON, and are expected to be used by the main application, e.g., to initialize XMON or to close it. Refer to the following XMON header file *xmon.h* for more details about these functions:

```
#include <xtensa/xmon.h>
```

19.2.1 _xmon_init()

```
int _xmon_init(char* gdbPkt, int gdbPktSize,
                 void(*xlog)(xmon_log_t type, const char* str));
```

The `_xmon_init()` function initializes and enables XMON. The application invokes `_xmon_init()` when it's ready to be debugged using XMON.

If debug exceptions or interrupts occur before `_xmon_init()` is called, XMON will try to ignore it: hardware breakpoints (IBREAK) and watchpoints (DBREAK) and stepping (ICOUNT) are disabled, interrupts are ignored, and breakpoints are skipped; there should not be any planted breakpoints.

The first two arguments, *gdbPkt* and *gdbPktSize*, are the start and size, respectively, of a buffer used for communication with GDB. The application allocates this buffer. The minimum required size is 100 bytes, and a reasonable default size for better performance is GDB_PKT_SIZE (4096) bytes.

The third argument, *xlog()*, is an optional pointer to a function called by XMON for all diagnostic output used to debug XMON itself, including error, logging, and tracing messages produced by XMON. It can be left NULL (0) if this functionality is not needed. See Section 19.5 for more details.

Note: Because XMON can be invoked and thus call this log function at any point in the application, care must be taken that the log function not call any non-reentrant function shared with the application. In particular, the C library `printf()` family of routines are notorious for being non-reentrant, and are best avoided. Thread-safety hooks used to provide re-entrance in a multi-threaded RTOS are insufficient here, as XMON ignores such mechanisms (XMON is not part of the RTOS and does not switch thread context).

XMON can also send certain diagnostic messages to GDB for display on the GDB console, independently of this log function. See Section 19.5 for details.

19.2.2 _xmon_close()

```
int _xmon_close(void);
```

The `_xmon_close()` function stops XMON debugging. After calling this function, XMON once again ignores debug exceptions (see Section 19.2.1), and the buffer provided to `_xmon_init()` can be deallocated. It is not necessary to call this function if an application is always debugged by XMON once debugging has started.

19.2.3 _xmon_log()

```
void _xmon_log(char app_log_en, char app_trace_en,
                 char gdb_log_en, char gdb_trace_en);
```

Enables the logging and tracing messages (see Section 19.5 for details) to the application and to the GDB. In total, the function provides four separate controls; two types of log messages, to either the application or the GDB.

19.2.4 _xmon_consoleOutput()

```
void _xmon_consoleOutput(char* message);
```

Sends a string to the GDB as a console message, using the XMON internal logging mechanism. This function is useful to observe the application operation and progress from the XMON attached GDB prompt.

19.2.5 _xmon_version()

```
char* _xmon_version(void);
```

Obtains the string containing the current XMON version.

19.3 User-provided Routines

XMON uses the following routines. These are expected to be provided by the user, *e.g.*, for a UART or other character stream interface used to communicate with GDB.

19.3.1 _xmon_in()

Receives characters from GDB as follows:

```
int _xmon_in(int block, unsigned char* buf);
```

The return value indicates the number of characters received in the array pointed to by `buf`. The current XMON version supports a single character input only (the return value is one). If no characters are available, the function might wait, or it must return zero if `block` is zero.

19.3.2 _xmon_out()

Outputs an array of characters to GDB as follows:

```
int _xmon_out(int len, unsigned char* buf);
```

The `len` argument indicates the number of characters to send out from the buffer pointed to `buf`.

19.3.3 _xmon_flush()

Flushes output characters as follows:

```
int _xmon_flush();
```

The application should send to GDB any internally buffered data (received with `xmon_out()`).

These are normally polling mode routines, as most interrupts are disabled during XMON execution.

The example file `xmon-main-serial.c` (see Section 19.7) provides an example implementation that works with supported FPGA boards using an UART port.

19.4 Invoking XMON

After initialization, all debug exceptions put XMON under GDB control. That is, XMON receives commands from GDB until GDB detaches. From the application side, it is necessary to establish the communication between GDB and XMON through a communication device (*e.g.*, UART). The device interrupt can simply execute a *break 1,15* instruction, which allows all requests from GDB to invoke XMON.

19.5 XMON Logging Messages

XMON can produce three types of messages: errors, logs, and traces. All three can be sent either to the application or to the GDB. The error messages are always enabled; logs are enabled by default while the traces need to be enabled either from the application or from the GDB using a monitor command.

Note that the most verbose messages (trace) are not compiled into the library. To receive trace messages as well, the application must use the XMON debug library (*libxmon-debug.a*).

19.5.1 XMON Logging via Application

The XMON initialization function (Section 19.2.1) specifies a handler for displaying XMON messages.The type of a message is defined using the following message type identifiers (`xmon_log_t` type): `XMON_LOG`, for logs, `XMON_ERR`, for errors, and `XMON_TRACE`, for traces. Enabling logs and traces can be done using the first two arguments to the`_xmon_log()`call.

19.5.2 XMON Logging via GDB

All three types of log messages are sent to the GDB using GDB asynchronous notification packets. They are all preceded with an "*XMON*" string. You can enable logs and traces using the next two arguments to the`_xmon_log()`call. In addition, they can be enabled directly from GDB using the following GDB monitor commands:

```
monitor {trace|log} {on|off}
```

These monitor commands enable or disable logging and tracing to GDB. Again, traces do not exist in the regular XMON library.

19.6 Requirements and Limitations

This section lists some requirements and limitations for debugging with XMON. Refer to Section 19.1 “XMON Overview”, where XMON is compared to XOCD for other limitations.

19.6.1 Avoiding XMON Recursion

XMON cannot debug itself, nor can it debug any function invoked in XMON. In particular, it cannot recover if a breakpoint is planted in some portions of XMON (for example in the XMON library), nor in some of the code invoked by XMON, specifically the following three HAL library functions (which an application might also use directly):

```
xthal_dcache_region_writeback
xthal_icache_region_invalidate
xthal_icache_sync
```

Avoid planting breakpoints in these functions (or in any XMON library function), or doing other debugger operations in them that result in planting breakpoints, such as the GDB `finish` and `until` commands.

Breakpoints may be planted in the XMON log handler function described in Section 19.2.1, or functions called by it, because XMON unplants breakpoints before invoking this function. However, such breakpoints will not be hit while XMON itself calls the log handler.

19.6.2 Writable Instruction Memory

XMON requires writable instruction memory to dynamically construct instructions that it executes to perform various tasks, including accesses to user, special, and TIE registers. For this, XMON pre-allocates some space within its code that is later used to dynamically create instructions. This space is labeled with a weak symbol (`_xmon_exec_instr`) so the application can provide a different space.

Following is an example of the XMON writable memory (48 bytes allocated) being placed in the IRAM0. Such code can be linked against the main application.

```
       .section .iram0.text, "ax"
       .align 4
_xmon_exec_instr:
       .space 48, 0
```

19.7 XMON Examples

XMON source files provide an example which uses the XMON library and enables UART on XTBSP targets (ML605 and KC705) boards as the GDB communication link. Sources are located in the following Xtensa Tools installation directory:

```
<xtensa_tools_root>/xtensa-elf/src/xmon
```

The program in *xmon-main-serial.c* is a simple counter loop that loops indefinitely and in each iteration increments the counter variable. The counter value is printed to the LCD and is also is sent to the GDB using XMON.The example also shows the implementation of the UART interrupt handler, required user-defined routines (see Section 19.3) and calls to different XMON provided routines (see Section 19.2). More details can be found in the aforementioned files.

Index