MIPS® EJTAG Specification

Document Number: MD00047
Revision 4.14
November 6, 2008

MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043-1353

Template: nB1.03, Built with tags: 2B

Table of Contents

List of Figures

List of Tables

Chapter 1

The EJTAG System

This specification describes the behavior and organization of on-chip EJTAG hardware resources as seen by software and by external agents. Software and firmware components of an EJTAG-based debugging environment are outside the scope of this document, as is the underlying physical implementation of EJTAG features.

This chapter contains the following sections:

- Section 1.1, "Introduction to EJTAG"
- Section 1.2, "Historical Perspective"
- Section 1.3, "EJTAG Capabilities"
- Section 1.4, "EJTAG Components and Options"
- Section 1.6, "EJTAG-Specific Coprocessor 0 Registers"
- Section 1.7, "Memory-Mapped EJTAG Registers"
- Section 1.8, "Memory-Mapped EJTAG Memory Segment"
- Section 1.9, "EJTAG Test Access Port Registers"
- Section 1.10, "The Implications of Multiprocessing and Multithreading for EJTAG"
- Section 1.11, "Related Documents"
- Section 1.12, "Notations and Conventions"

For comments or questions on the EJTAG Architecture or this document, send Email to support@mips.com.

1.1 Introduction to EJTAG

EJTAG is a hardware/software subsystem that provides comprehensive debugging and performance tuning capabilities to MIPS® microprocessors and to system-on-a-chip components having MIPS processor cores. It exploits the infrastructure provided by the IEEE 1149.1 JTAG Test Access Port (TAP) standard to provide an external interface, and extends the MIPS instruction set and privileged resource architectures to provide a standard software architecture for integrated system debugging.

1.2 Historical Perspective

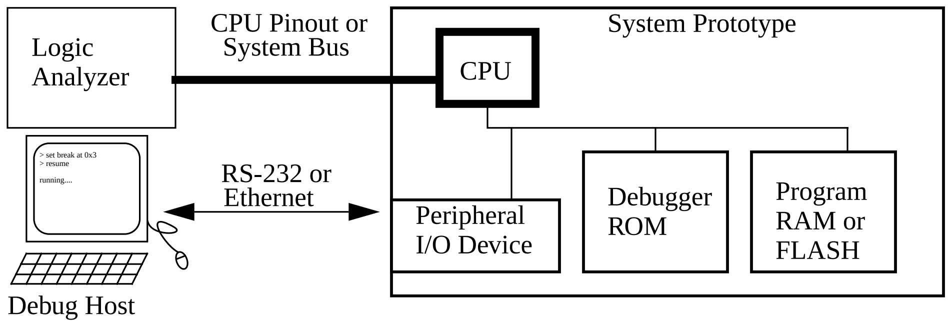
Emulating and debugging embedded hardware and software in a real-world environment remains one of the most difficult tasks facing designers of embedded systems today. Embedded microprocessor cores are growing more complex, have increasingly higher performance, and use larger software programs than ever before. To meet the challenge, embedded systems engineers and programmers must have advanced tools to perform the required levels of in-circuit emulation and debugging.

The MIPS architecture has historically provided a set of primitives for debugging software and systems that is consistent with the "RISC" philosophy of integrated hardware/software architecture, providing functionality at a minimum cost in silicon. The base philosophy of integrated MIPS32®/MIPS64® Instruction Set Architecture (ISA) and MIPS16e™ Application Specific Extension (ASE), includes:

- A breakpoint instruction, BREAK, whose execution causes a specific exception.
- A set of trap instructions, whose execution causes a specific exception when certain register value criteria are satisfied.
- A pair of optional Watch registers that can be programmed to cause a specific exception on a load, store, or instruction fetch access to a specific 64-bit doubleword in virtual memory.
- An optional TLB-based MMU that can be programmed to trap on any access, or more specifically, on any store to a page of memory.

All of these mechanisms assume software support in the form of an operating system, or at least a software monitor, that can modify program memory to insert breakpoints, manipulate the system coprocessor to set watchpoints and change virtual memory page protection, handle the exceptions produced, and communicate with a user. Additional external hardware tools can supplement these basic mechanisms, such as logic analyzers and in-circuit emulators (ICEs) for additional control and information about program execution. Figure 1.1 shows a possible setup for the debug of an embedded system.

Figure 1.1 Setup of Debug System without EJTAG



While this model of debug works well for many sorts of system, it has the following shortcomings when the system to be debugged is a highly-integrated design:

- System-On-a-Chip (SOC) component design no longer provides an external interface to the processor pin-out or system bus, making the use of logic analyzers and ICEs difficult to impossible.
- Debugging based on software breakpoints or the insertion of trap-on-condition instructions assumes that programs reside in RAM. It is impractical for fully ROM-based systems and assumes support in the O/S for these techniques.
- For consumer electronic applications, a communication port like Ethernet or RS-232 serves no purpose beyond software debug and adds disproportionately to the cost and size of the design.

- Similarly, the ROM necessary to support a debug software monitor on a consumer electronic application could add unacceptable costs.

One alternative to ICE is a specially-packaged device that is a bond-out of the chip. But this solution has the disadvantage of adding to overall product development cost. It also adds the extra requirement of a specially-designed PCB that is needed to access the signals available only on the development chip.

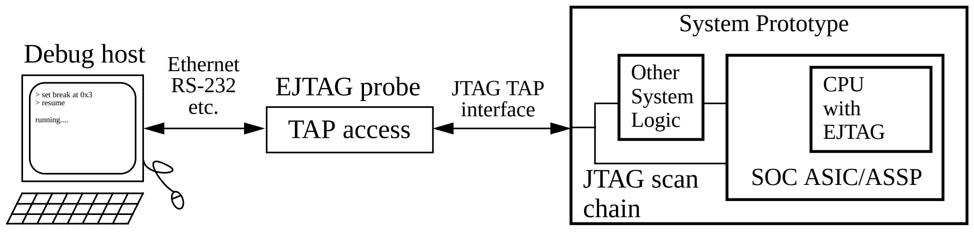
On-Chip Debug (OCD) provides a solution for all these issues, and the EJTAG Debug Solution defines an advanced and scalable feature-set for OCD that allows debugging while executing CPU code at full speed.

One could say that OCD puts the ICE functionality on the chip. Although OCD does add a little extra die area for features that are only required during development, the die area is minimal. More importantly, with development time and overall time-to-market becoming increasingly critical, the trade-off between die area and time seems reasonable.

Having the debug solution on-chip also makes it possible to use it for software upgrades, and field testing, and for diagnostics in the final product.

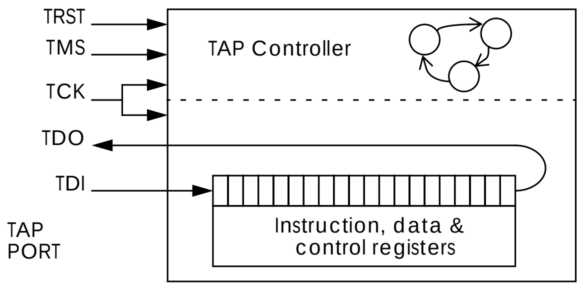
EJTAG supplements the MIPS Architecture in dealing with these problems. A processor or system-on-a-chip implementing EJTAG can be tied into a JTAG scan chain and comprehensively debugged using an external EJTAG probe connected to the system's JTAG TAP interface, as shown in Figure 1.2.

Figure 1.2 Setup of Debug System with EJTAG



EJTAG uses the five-pin interface defined in IEEE 1149.1 JTAG, which forms the Test Access Port (TAP). The five pins (TRST, TCK, TMS, TDI, and TDO) can be reused to limit pin count if the TAP is on-chip for some other purpose.

Figure 1.3 Test Access Port (TAP) to Internal Connections



This EJTAG interface through the TAP is a serial communications channel with frequencies up to 40 MHz on TCK. The TAP Controller uses the TMS pin, which determines if instruction or data registers should be accessed in the shift path between TDI and TDO. The TRST signal is used for reset of the TAP.

A number of TAP instructions are defined in EJTAG that allow access to corresponding EJTAG registers, as listed in Table 1.1.

Table 1.1 EJTAG TAP Instructions

EJTAG Instructions	Description of Register Usage
IDCODE	Device Identification Register with manufacturer, part number, and version ID for the specific chip.
IMPCODE	Implementation Register indicating implemented EJTAG features in this specific chip.
ADDRESS	EJTAG Address Register used to access the on-chip address bus.
DATA	EJTAG Data Register used to access the on-chip data bus.
CONTROL	EJTAG Control Register used for setup and status information.
ALL	Access to EJTAG Address, Data and Control registers in one chain.
EJTAGBOOT	Causes processor to take a debug exception after reset.
NORMALBOOT	Causes processor to execute the reset handler after reset.
FASTDATA	Access to the Data and FastData registers.
TCBCONTROLA	Access to the control register *TCBControlA* in the Trace Control Block (TCB).
TCBCONTROLB	Access to the other control register *TCBControlB* in the TCB.
TCBDATA	Provides access to the registers specified by the $\text{TCBCONTROLB}_{\text{REG}}$ field.
TCBCONTROLC	Access to another control register *TCBControlC* in the TCB.
PCSAMPLE	Access the PCsample register.
TCBCONTROLD	Access to another control register TCBControlD in the TCB.
TCBCONTROLE	Access to another control register TCBControlE in the TCB.
BYPASS	One-bit register with no operation.

The size of the EJTAG Address and Data Registers depends on the specific implementation, but usually they are at least 32 bits. The size of the Device ID, Implementation, and EJTAG Control Registers is 32 bits; these registers allow the user to do debug setup and provide important status information during the debug session. For exact descriptions and size of these registers see 7.4 "Instruction Register and Special Instructions" on page 123.

1.3 EJTAG Capabilities

1.3.1 Debug Exception and Debug Mode

To allow inspection of the CPU state at any time in the execution flow, a debug exception with priority over all other exceptions is introduced.

When a debug exception occurs, the CPU goes into Debug Mode, a special mode with no restrictions on access to coprocessors, memory areas, etc., and where usual exceptions like address error and interrupt are masked.

The debug exception handler is executed in Debug Mode and provided by the debug system. It can be executed from the probe through a processor access, or may also reside in the application code if the developer chooses to use a debug task in the application.

An overall requirement is that debugging be non-intrusive to the application so execution of the application can be continued after the needed debug operations. However, loss of real-time operation is inevitable when the debug exception handler is executed. The system designer may chose to indicate debug mode by a signal to certain hardware modules to freeze them when executing the debug exception handler.

EJTAG provides a standard debug I/O interface, enabling the use of traditional MIPS debug facilities on system-on-a-chip components. In addition, EJTAG provides the following new capabilities for software and system debug.

1.3.2 Off-board EJTAG Memory

EJTAG allows a MIPS processor in Debug Mode to reference instructions or data that are not resident on the system under test. This EJTAG memory is mapped to the processor as if it were virtual memory in the kseg3 segment, and references to it are converted into transactions on the TAP interface. Both instructions and data can be accessed in EJTAG memory, which allows debugging of systems without requiring the presence of a ROM monitor or debugger scratchpad RAM. It also provides a communications channel between debug software executing on the processor and an external debugging agent.

The EJTAG probe polls the EJTAG Control Register through the TAP, and a bit in this register indicates when a processor access is pending. The physical address of the transaction is then available in the EJTAG Address Register, and the transaction size and read/write indication are available in the EJTAG Control Register. The EJTAG Data Register is then accessed either to get data from a write or to provide data for a read. Finally the EJTAG Control Register is updated to indicate that the processor access is done.

Going through this sequence requires on order of 200 TCK periods for access to 32-bit address and data registers. With a 40 MHz TCK, the access time is in the range of 5 µs, resulting in a bandwidth in the range of 800 KB/s for instruction or data transfers. However, the servicing may be optimized for instruction stuffing since the address depends on the provided instructions and could thus be predicted to some extent. In addition, the FASTDATA feature (see Section 7.4.3 "FASTDATA Instruction") of the TAP controller permits fast download or upload of data between target memory and debug memory.

1.3.3 Debug Breakpoint Instruction

EJTAG introduces a new breakpoint instruction, SDBBP, which differs from the MIPS32 and MIPS64 BREAK instruction in that the resulting exception, like the single-step and hardware breakpoint debug exceptions described below, places the processor in Debug Mode and can fetch its associated handler code from EJTAG memory.

1.3.4 Hardware Breakpoints

EJTAG defines various types of hardware breakpoints for interrupting the CPU at certain transactions on the CPU buses. The debug exception happens before the bus transaction causing the exception alters any memory or CPU state, e.g., a fetched instruction with a break is not executed, or a data load/store transaction is not allowed to change the register file or the memory.

Hardware breaks on instructions have the advantage over software debug breaks in that it is possible to set them in any address area. Furthermore, if memory cannot be altered by inserting SDBBP codes, the hardware breaks can still be used. Hardware data breakpoints allow breaks on load/store operations.

EJTAG implements two kinds of simple breaks:

- Instruction breaks, in which a break may be set on an instruction fetch from a specific virtual address

- Data breaks, in which a break may be set on a load/store reference from a specific virtual address, which additionally can be qualified by a data value.

There may be up to 15 break channels of each type implemented, and each break channel may be programmed with address, address mask, ASID, and reference type.

EJTAG specification 4.00 and above also define complex breakpoints. There are many different types of complex breakpoints defined the complex break chapter. Like the simple breaks, the complex breaks can cause a trigger signal that can be used to enable or disable tracing via the MIPS PDtrace architecture.

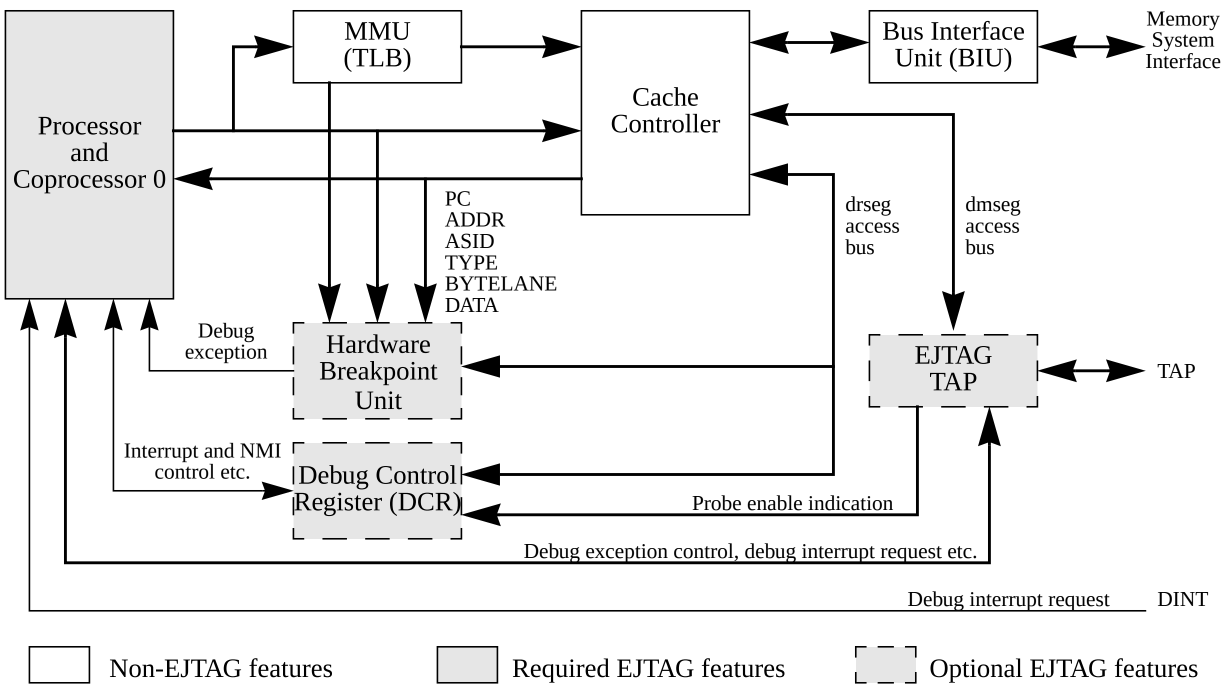
1.3.5 Single-Step Execution

EJTAG provides support for single-step execution of programs and operating systems, without requiring that the code reside in RAM.

1.4 EJTAG Components and Options

EJTAG hardware support consists of several distinct components: extensions to the MIPS processor core, the EJTAG Test Access Port, the Debug Control Register, and the Hardware Breakpoint Unit. Figure 1.4 shows the relationship between these components in an EJTAG implementation. Some components and features are optional, and are implemented based on the needs of an implementation.

Figure 1.4 Simplified Block Diagram of EJTAG Components



1.4.1 EJTAG Processor Core Extensions

A MIPS processor or core supporting EJTAG must support EJTAG-specific instructions, additional system coprocessor (CP0) registers and vectoring to Debug Exceptions, which puts the processor in a special Debug Mode of execution, as described in Chapter 6, “EJTAG Processor Core Extensions” on page 81.

EJTAG processor core extensions are required in any EJTAG implementation, with the following implementation-dependent options:

- The single-step execution feature is optional. The presence or absence of single step execution capability is indicated to debug software via the CP0 Debug register.
- The debug interrupt request from the TAP via the DINT probe signal or through an implementation-dependent internal signal is optional.
- The Test Access Port (TAP) is optional.
- The Hardware Breakpoint Unit (HBU) is optional. Note that it is required if the CBT is implemented.
- The Complex Break and Trigger (CBT) block is optional.
- The Debug Control Register (DCR) is optional. Note that it is required if either the HBU or the CBT is implemented.
- The PC Sampling feature of EJTAG is optional.

1.4.2 EJTAG Test Access Port

The EJTAG Test Access Port (TAP) provides a standard TAP interface to the EJTAG system. It is necessary for all TAP-based EJTAG capabilities for host-based debugging and processor access to external debug memory.

The TAP is optional. Implementation without a TAP implicitly disallows the EJTAG memory and TAP system access capabilities, but provides the remaining EJTAG services (Debug Mode, single-step, software and hardware breakpoints) while executing from RAM or ROM. Refer to Chapter 7, “EJTAG Test Access Port” on page 119 for more information on the TAP.

Implementation without a TAP also disallows the PC Sampling feature.

The presence or absence of off-board EJTAG memory is indicated to debug software via the Debug Control Register.

1.4.3 Debug Control Register

The Debug Control Register (DCR) is a memory-mapped register that can be implemented as part of either the processor core or an external logic block. It indicates the availability and status of EJTAG features. The memory-mapped region containing the DCR is available to software only in Debug Mode.

Implementation of the DCR is optional, but the DCR must be implemented if either the EJTAG TAP or EJTAG hardware breakpoints are implemented. The presence or absence of the DCR is indicated in the CP0 Debug register. Refer to Chapter 2, “Debug Control Register” on page 29 for more information on the DCR.

1.4.4 Hardware Breakpoint Unit

The Hardware Breakpoint Unit implements memory-mapped registers that control the instruction and data hardware breakpoints. The memory-mapped region containing the hardware breakpoint registers is accessible to software only in Debug Mode.

EJTAG hardware breakpoint support, as described in Chapter 3, “Hardware Breakpoints” on page 33, is optional, and can be implemented with the following functionality:

- From zero to 15 independent instruction hardware breakpoints
- From zero to 15 independent data hardware breakpoints
- Breakpoint address comparisons for instruction and data hardware breakpoints optionally qualified with a comparison of the MMU ASID
- Data hardware breakpoints optionally qualified with a data value comparison
- The sense of the data value qualifier can be inverted, that is, when the store data for example does NOT match the specified value in the data break register. This is an optional functionality whose presence is indicated by a bit (15) in the DCR register. This feature is defined in revision 4.00 and above.
- Debug logic can optionally save the load data value in a specified drseg address register for software replay of the exception-causing load instruction. This is needed to preserve the load data value in situations where the data was obtained not from non-volatile memory but from say a FIFO or an I/O register. Whether or not this feature is implemented is indicated by a bit (14) in the DCR register. This feature is defined in revision 4.00 and above.

The presence or absence of hardware breakpoint capability is indicated to debug software in the DCR.

The number of breakpoints and the availability of optional qualifiers is indicated to debug software in the instruction and data breakpoint status registers.

1.5 Complex Breakpoint and Trigger (CBT) Block

The presence or absence of this optional block is indicated by a bit (10) in the DCR register. Each of the listed features of this block is optional and the presence or absence of this feature is indicated by bits in the CBTcontrol register which is a drseg address-mapped register at address 0x8000:

- Pass Counters - each break channel, instruction, data, or complex has a counter associated with it that enables a breakpoint to only be taken after the address/value condition has been met a certain number of times.
- Ability to support 0 to 15 ‘tuples’ - breakpoints that only fire when both instruction and data conditions match on a single instruction.
- Qualified Instruction breakpoints - breakpoints that can be enabled and disabled based on the state of a data breakpoint condition which can be used to only match on instructions executed in a certain process.
- Primed breakpoints - breakpoints that are only enabled once another breakpoint has occurred which allows breaking on a simple sequences of events. It is an implementation choice as to how many priming conditions are supported for each break, upto 16 priming conditions are possible. Note that the default priming condition is the simple break, that is, no priming condition.

- Stopwatch timer - a counter that can be configured to start or stop based on specific instruction breakpoints. It is an implementation choice which breakpoints are used to start and stop the stopwatch timer. Upto two such pairs may be supported.

1.6 EJTAG-Specific Coprocessor 0 Registers

This section summarizes the registers and special memory that are used for the EJTAG debug solution. More detailed information regarding mandatory and optional registers and memory locations is described in the relevant chapter.

Table 1.2 summarizes the Coprocessor 0 (CP0) registers. These registers are accessible by the debug software executed on the processor; they provide debug control and status information. General information about the debug CP0 registers is found in 6.7 "EJTAG Coprocessor 0 Registers" on page 104.

Table 1.2 Overview of Coprocessor 0 Registers for EJTAG

Register Name	Register Mnemonic	Functional Description	Reference
Debug	Debug	Debug indications and controls for the processor, including information about recent debug exception.	See Section 6.7.1 on page 104
Debug Exception Program Counter	DEPC	Program counter at last debug exception or exception in Debug Mode.	See Section 6.7.3 on page 113
Debug Exception Save	DESAVE	Scratchpad register available for the debug handler.	See Section 6.7.4 on page 114

1.7 Memory-Mapped EJTAG Registers

The memory-mapped EJTAG registers are located in the debug register segment (drseg), which is a sub-segment of the debug segment (dseg). They are accessible by debug software when the processor is executing in Debug Mode. These registers provide both miscellaneous debug control and control of hardware breakpoints. General information about the debug segment and registers is found in Section 6.2.2 on page 82 and Section 6.2.2.2 on page 86.

1.7.1 Debug Control Register

Table 1.3 summarizes the Debug Control Register (DCR), which provides miscellaneous debug control.

Table 1.3 Overview of Debug Control Register as Memory-Mapped Register for EJTAG

Register Name	Register Mnemonic	Functional Description	Reference
Debug Control Register	DCR	Indicates available EJTAG memory, and controls enabling and disabling of interrupts and NMI in Non-Debug Mode.	See Chapter 2, "Debug Control Register" on page 29

1.7.2 Instruction Hardware Breakpoint Registers

Table 1.4 summarizes the instruction hardware breakpoint registers, which are controlled through a number of memory-mapped registers. Certain registers are provided for each implemented instruction hardware breakpoint, as indicated with an "n". General information about the instruction hardware breakpoint registers is found in Section 3.6 on page 46.

Table 1.4 Overview of Instruction Hardware Breakpoint Registers

Register Name	Register Mnemonic	Functional Description	Reference
Instruction Breakpoint Status	IBS	Indicates number of instruction hardware breakpoints and status on a previous match.	See Section 3.6.1 on page 47
Instruction Breakpoint Address (n)	IBAn	Address to compare for breakpoint n.	See Section 3.6.2 on page 48
Instruction Breakpoint Address Mask (n)	IBMn	Mask for address comparison for breakpoint n.	See Section 3.6.3 on page 48
Instruction Breakpoint ASID (n)	IBASIDn	ASID value to compare for breakpoint n.	See Section 3.6.4 on page 49
Instruction Breakpoint Control (n)	IBCn	Control of breakpoint n: comparison of ASID and generated event on match.	See Section 3.6.5 on page 50

1.7.3 Data Hardware Breakpoint Registers

Table 1.5 summarizes the data hardware breakpoint registers, which are controlled as a number of memory-mapped registers. Certain registers are provided for each implemented data hardware breakpoint, as indicated with an "n". General information about the data hardware breakpoint registers is found in Section 3.7 on page 51.

Table 1.5 Overview of Data Hardware Breakpoint Registers

Register Name	Register Mnemonic	Functional Description	Reference
Data Breakpoint Status	DBS	Indicates number of data hardware breakpoints and status on a previous match.	See Section 3.7.1 on page 52
Data Breakpoint Address (n)	DBAn	Address to compare for breakpoint n.	See Section 3.7.2 on page 53
Data Breakpoint Address Mask (n)	DBMn	Mask for address comparison for breakpoint n.	See Section 3.7.3 on page 54
Data Breakpoint ASID (n)	DBASIDn	ASID value to compare for breakpoint n.	See Section 3.7.4 on page 54
Data Breakpoint Control (n)	DBCn	Control of breakpoint n: match on load/store, data bytes, access to data bytes, comparison of ASID, and generated event on match.	See Section 3.7.5 on page 55
Data Breakpoint Value (n)	DBVn	Data value to match for breakpoint n.	See Section 3.7.6 "Data Breakpoint Value n (DBVn) Register"

1.8 Memory-Mapped EJTAG Memory Segment

The processor’s memory-mapped EJTAG memory is located in the debug memory segment (dmseg), which is a sub-segment of the debug segment (dseg). It is accessible by debug software when the processor is executing in Debug Mode. The EJTAG probe handles all accesses to this segment through the Test Access Port (TAP), whereby the processor has access to dedicated debug memory even if no debug memory was originally located in the system. General information about the debug segment and memory is found in Section 6.2.2 on page 82 and Section 6.2.2.1 on page 85.

1.9 EJTAG Test Access Port Registers

The probe accesses EJTAG Test Access Port (TAP) registers (shown in Table 1.6) through the TAP, so the processor can not access these registers. These registers allow specific control of the target processor through the TAP. General information about the TAP registers is found in Section 7.5 on page 126.

Table 1.6 Overview of Test Access Port Registers

Register Name	Register Mnemonic	Functional Description	Reference
Device ID	(none)	Identifies device and accessed processor in the device.	See Section 7.5.1 on page 127
Implementation	(none)	Identifies main debug features implemented and accessible through the TAP.	See Section 7.5.2 on page 128
Data	(none)	Data register for processor accesses used to support the EJTAG memory.	See Section 7.5.3 on page 130
Address	(none)	Address register for processor access used to support the EJTAG memory.	See Section 7.5.4 on page 132
EJTAG Control	ECR	Control register for most EJTAG features used through the TAP.	See Section 7.5.5 on page 133
Bypass	(none)	Provides a one-bit shift path through the TAP.	See Section 7.5.8 on page 141
Fastdata	(none)	Provides a one-bit tag in front of the data register to capture the processor access pending bit for fast data transfer.	See Section 7.5.8 on page 141
TCBContolA	(none)	Used by the Trace Control Block to hold control bits for tracing.	See the PDtrace and TCB specification document
TCBControlB	(none)	Used by the Trace Control Block to hold control bits for tracing.	See the PDtrace and TCB specification document
TCBData	(none)	Used by the Trace Control Block to access data from on-chip trace memory if present	See the PDtrace and TCB specification document
TCBControlC	(none)	Used by the Trace Control Block to hold control bits for tracing	See the PDtrace and TCB specification document

Table 1.6 Overview of Test Access Port Registers (Continued)

Register Name	Register Mnemonic	Functional Description	Reference
PCsample	(none)	Used by the PC Sampling logic to write out the PC sample and associated information	See Section 7.5.7 on page 141 and Chapter 5, “PC Sampling” on page 79.
TCBControlD	(none)	Used by the Trace Control Block to hold control bits for tracing	See the PDtrace and TCB specification document
TCBControlE	(none)	Used by the Trace Control Block to hold control bits for tracing	See the PDtrace and TCB specification document

1.10 The Implications of Multiprocessing and Multithreading for EJTAG

The MIPS® MT ASE allows a processor to implement multiple VPEs (Virtual Processing Elements). Theoretically, as far as applications are concerned, this view of the hardware (which must be supported by system software), is no different from that where there are multiple physical processors present. MIPS MT also allows multiple thread contexts within a VPE. See the MIPS MT specification for details.

EJTAG visibility is on a per-VPE or per-processor basis. That is, each debug unit implemented in the system exposes a TAP controller to the external probe hardware. The probe software must be aware of the number of daisy-chained debug units and their order so that it can communicate correctly to the right debug unit.

Note that by the MIPS MT ASE specification, an implementation with multiple VPEs and hence multiple debug units, most of the EJTAG hardware is physically not shared between the VPEs. For example, each VPE has its own copy of the Debug Register, Debug Control Register, TAP controller, and TAP registers. But the hardware breakpoint registers may either be shared or not shared by the VPEs. The TAP controllers are daisy-chained.

The other sections in this document that describe changes for the MIPS MT ASE are:

- Debug Exception in the presence of MIPS MT (see Section 6.2 on page 82).
- Single-Step control bit in the Debug register (see Section 6.7 on page 104 and Section 6.3.8 on page 96).
- Modifications to the Instruction and Data breakpoints matching conditions (see Section 3.3 on page 36).
- Modifications to the Instruction and Data Hardware Breakpoint registers for MIPS MT (see Section 3.6.5 on page 50, Section 3.7.4 on page 54, and Section 3.7.5 on page 55).
- Modification to indicate whether the Instruction and Data Hardware Breakpoints are shared or not shared across the VPEs (see Section 3.6.1 on page 47 and Section 3.7.1 on page 52).
- A bit added to the DCR (VPED), to indicate whether the current VPE is disabled or enabled.
- A bit added to the Debug register to allow MIPS MT thread contexts (TCs) to be taken off-line during debug (see Section 6.7.1 on page 104).

1.11 Related Documents

The following documents are useful in understanding this specification.

- IEEE Std. 1149.1-1990, *IEEE Standard Test Access Port and Boundary-Scan Architecture*
- MIPS32® Architecture for Programmers, Volumes I-IV
- MIPS64® Architecture for Programmers, Volumes I-IV
- The PDtrace™ Interface and Trace Control Block Specification (MD00439)
- MIPS32® Architecture for Programmers Volume IV-f: The MIPS® MT Application-Specific Extension to the MIPS32® Architecture (MD00378)
- The iFlowtrace™ Architecture Specification (MD00526)

1.12 Notations and Conventions

This section defines notations and conventions that are used throughout this document.

1.12.1 Compliance

Throughout this document, compliance levels are indicated for specific features. Features are defined as Required, Optional, or Recommended.

Features defined as *required* are required of all processors claiming compatibility with the EJTAG architecture.

Features defined as *optional* provide a standardization that might or might not be appropriate for a particular EJTAG implementation. If such a feature is implemented, it must be implemented as described in this document for a processor to claim compatibility with the EJTAG architecture.

In some cases, there are features within features that have different levels of compliance. For example, if there is an optional field within a required register, the register must be implemented, but the field does not have to be implemented, depending on the needs of the implementation. Similarly, if there is a required field within an optional register, if the register is implemented, it must have the specified field.

Features defined as *recommended* should be implemented unless there is an overriding need not to do so.

1.12.2 UNPREDICTABLE and UNDEFINED Operations

These definitions of UNPREDICTABLE and UNDEFINED are similar to the descriptions in the MIPS32 and MIPS64 specifications. They are included here for those readers who are not familiar with these documents.

The terms UNPREDICTABLE and UNDEFINED describe the behavior of the processor in certain cases. UNDEFINED behavior or operations can occur only as the result of executing instructions in a privileged mode (in Kernel Mode or Debug Mode, or with the CP0 usable bit set in the Status register). Unprivileged software can never cause UNDEFINED behavior or operations. Conversely, both privileged and unprivileged software can cause UNPREDICTABLE results or operations.

1.12.2.1 UNPREDICTABLE

UNPREDICTABLE results can vary from implementation to implementation, instruction to instruction, or as a function of time in the same implementation or instruction. Software can never depend on results that are UNPREDICTABLE. An UNPREDICTABLE operation might or might not cause a result to be generated. If it does generate a result, the result is UNPREDICTABLE. UNPREDICTABLE operations can cause arbitrary exceptions.

UNPREDICTABLE results or operations have several implementation restrictions:

- UNPREDICTABLE results must not depend on any data source (memory or internal state) that is inaccessible in the current processor mode.
- UNPREDICTABLE operations must not read, write, or modify the contents of memory or an internal state that is inaccessible in the current processor mode. For example, UNPREDICTABLE operations executed in User Mode must not access memory or internal state that is only accessible in Kernel Mode, Debug Mode, or in another process.
- UNPREDICTABLE operations must not halt or hang the processor.

1.12.2.2 UNDEFINED

UNDEFINED operations or behavior can vary from implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. UNDEFINED operations or behavior can vary from nothing to creating an environment in which execution can no longer continue. UNDEFINED operations or behavior can cause data loss.

UNDEFINED operations or behavior must not cause the processor to hang (that is, enter a state from which there is no exit other than powering down the processor). The assertion of any reset signal must restore operation to a deterministic state.

1.12.3 Register Field Notations

Table 1.7 defines the notations used to describe the read/write properties of the registers in this document. The notations below are similar to those in the MIPS32 and MIPS64 specifications, with addition of R/W0 and R/W1.

Table 1.7 Register Field Notations

Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W	A field in which all bits are readable and writable by software and potentially by hardware. Hardware updates of this field are visible by software reads. Software updates of this field are visible by hardware reads. If the Reset State of this field is “Undefined”, either software or hardware must initialize the value before the first read will return a predictable value. This operation should not be confused with the formal definition of UNDEFINED behavior.	
R/W0	Similar to the R/W interpretation, except a software write of value 1 to this bit is ignored.	
R/W1	Similar to the R/W interpretation, except a software write of value 0 to this bit is ignored.	

Table 1.7 Register Field Notations

Read/Write Notation	Hardware Interpretation	Software Interpretation
R	A field that is either static or updated only by hardware. If the Reset State of this field is either "0" or "Preset", hardware initializes this field to zero or to the appropriate state, respectively, on power-up. If the Reset State of this field is "Undefined", hardware updates this field only under those conditions specified in the description of the field.	A field to which the value written by software is ignored by hardware. Software can write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware. If the Reset State of this field is "Undefined", software reads of this field result in an UNPREDICTABLE value except after a hardware update done under the conditions specified in the description of the field.
0	A field that hardware does not update, and for which hardware can assume a zero value.	A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in UNDEFINED behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zeros. If the Reset State of this field is "Undefined", software must write this field with zero before it is guaranteed to read as zero.

1.12.4 Value Notations

The following conventions are used for numeric values in this document:

- Decimal values are written as standard base 10 numbers.
- Hexadecimal values are prefaced with "0x".
- Binary numbers are appended with "$_2$".

For example, the following numbers are equivalent: 13 = = 0xD = = 1101_2.

1.12.5 Address Notations

Except where addresses are obviously 32 bits by context (as for a R3k privileged environment), addresses in this document are shown as 64 bits. For 32-bit implementations, ignore the upper 32 bits of the address.

Addresses (ADDR) are usually marked in hexadecimal notation as 0xADDR.

Chapter 2

Debug Control Register

Compliance Level: Optional, but requires EJTAG processor core extensions. If this register is not implemented then other features that depend on bits in this register behave as if these bits are present and have the reset value.

The Debug Control Register (DCR) controls and provides information about debug issues. The width of the register is 32 bits for 32-bit processors, and 64 bits for 64-bit processors. The DCR is located in the drseg segment at offset 0x0000.

The Debug Control Register (DCR) provides the following key features:

- Interrupt and NMI control when in Non-Debug Mode
- NMI pending indication
- Availability indicator of instruction and data hardware breakpoints
- Availability of the PC sample feature and the sample period

For EJTAG features, there are no difference between a reset and a soft reset occurring to the processor; they behave identically in both Debug Mode and Non-Debug Mode. References to reset in the following therefore refers to both reset (hard reset) and soft reset.

The DataBrk and InstBrk bits within the DCR indicate the types of hardware breakpoints implemented. Debug software is expected to read hardware breakpoint registers for additional information on the number of implemented breakpoints. Refer to Chapter 3, “Hardware Breakpoints” on page 33 for descriptions of the hardware breakpoint registers.

Hardware and software interrupts can be disabled in Non-Debug Mode using the DCR’s IntE bit. This bit is a global interrupt enable used along with several other interrupt enables that enable specific mechanisms. The NMI interrupt can be disabled in Non-Debug Mode using the DCR’s NMIE bit; a pending NMI is indicated through the NMIpend bit. Pending interrupts are indicated in the Cause register, and pending NMIs are indicated in the DCR register NMIpend bit, even when disabled. Hardware and software interrupts and NMIs are always disabled in Debug Mode. See Section 6.5 on page 102 for more information.

The optional SRstE bit allows masking of soft resets. A soft reset can be applied to the system based on different events, referred to as sources. It is implementation dependent which soft reset sources in a system can be masked by the SRstE bit. Soft reset masking can be applied to a soft reset source only if that source can be efficiently masked in the system. The result is no reset at all for any part of the system, if masked. If only a partial soft reset is possible, then that soft reset source is not to be masked, because a “half” soft reset might cause the system to fail or hang without warning. There is no automatic indication of whether the SRstE bit is effective, so the user must consult system documentation.

The ProbEn bit reflects the state of the ProbEn bit from the EJTAG Control register (ECR). Through this bit, the probe can indicate to the debug software running on the CPU if it expects to service dmseg segment accesses. See Section 7.5.5 on page 133 for more information.

Figure 2.1 shows the format of the DCR register; Table 2.1 describes the DCR register fields. The reset values in Table 2.1 take effect on both hard resets and soft resets.

Figure 2.1 DCR Register Format

	31 30	29	28 18	17	16	15	14	13 11	10	9	8 6	5	4	3	2	1	0
32-bit Processor	0	ENM	0	DataBrk	InstBrk	IVM	DVM	0	CBT	PCS	PCR	PCSe	IntE	NMIE	NMIpend	SRstE	ProbEn

	63 30	29	28 18	17	16	15	14	13 11	10	9	8 6	5	4	3	2	1	0
64-bit Processor	0	ENM	0	DataBrk	InstBrk	IVM	DVM	0	CBT	PCS	PCR	PCSe	IntE	NMIE	NMIpend	SRstE	ProbEn

Table 2.1 DCR Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
ENM	29	Endianess in which the processor is running in kernel and Debug Mode: **Encoding** | **Meaning** 0 | Little endian 1 | Big endian	R	Preset	Required
DataBrk	17	Indicates if data hardware breakpoint is implemented: **Encoding** | **Meaning** 0 | No data hardware breakpoint implemented 1 | Data hardware breakpoint implemented	R	Preset	Required
InstBrk	16	Indicates if instruction hardware breakpoint is implemented: **Encoding** | **Meaning** 0 | No instruction hardware breakpoint implemented 1 | Instruction hardware breakpoint implemented	R	Preset	Required
IVM	15	Indicates if inverted data value match on data hardware breakpoints is implemented: **Encoding** | **Meaning** 0 | No inverted data value match on data hardware breakpoints implemented 1 | Inverted data value match on data hardware breakpoints implemented	R	Preset	Required

Table 2.1 DCR Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
DVM	14	Indicates if a data value store on a data value breakpoint match is implemented: **Encoding** / **Meaning** 0: No data value store on a data value breakpoint match implemented 1: Data value store on a data value break-point match implemented	R	Preset	Required
CBT	10	Indicates if complex breakpoint block is implemented: **Encoding** / **Meaning** 0: No complex breakpoint block imple-mented 1: Complex breakpoint block imple-mented	R	Preset	Required
PCS	9	Indicates if the PC Sampling feature is implemented. **Encoding** / **Meaning** 0: No PC Sampling implemented 1: PC Sampling implemented	R	Preset	Required
PCR	8:6	PC Sampling rate. Values 0 to 7 map to values 2^5 to 2^{12} cycles, respectively. That is, a PC sample is written out every 32, 64, 128, 256, 512, 1024, 2048, or 4096 cycles respectively. The external probe or software is allowed to set this value to the desired sample rate.	R/W	0	Required if PCS is 1
PCSe	5	If the PC sampling feature is implemented, then indi-cates whether PC sampling is initiated or not. That is, a value of 0 indicates that PC sampling is not enabled and when the bit value is 1, then PC sampling is enabled and the counters are operational.	R/W	0	Required if PCS is 1
IntE	4	Hardware and software interrupt enable for Non-Debug Mode, in conjunction with other disable mechanisms: **Encoding** / **Meaning** 0: Interrupt disabled 1: Interrupt enabled depending on other enabling mechanisms	R/W	1	Required
NMIE	3	Non-Maskable Interrupt (NMI) enable for Non-Debug Mode: **Encoding** / **Meaning** 0: NMI disabled 1: NMI enabled	R/W	1	Required

Table 2.1 DCR Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
NMIpend	2	Indication for pending NMI: **Encoding** / **Meaning** 0: No NMI pending 1: NMI pending	R	0	Required
SRstE	1	Controls soft reset enable: **Encoding** / **Meaning** 0: Soft reset masked for soft reset sources dependent on implementation 1: Soft reset is fully enabled Bit is read-only (R) and reads as zero if not implemented.	R/W	1	Optional
ProbEn	0	Indicates value of the ProbEn value in the ECR register: **Encoding** / **Meaning** 0: No access should occur to the dmseg segment 1: Probe services accesses to the dmseg segment Bit is read-only (R) and reads as zero if not implemented.	R	Same value as ProbEn in ECR	Required if EJTAG TAP is present, otherwise not implemented
0	MSB:30, 28:18, 13:11	Must be written as zeros; return zeros on reads.	0	0	Reserved

Chapter 3

Hardware Breakpoints

This chapter describes the optional instruction and data hardware breakpoints. It contains the following sections:

The general description in this chapter covers processors with R4k privileged environments. Differences for processors with R3k privileged environments are described in Appendix A, "Differences for R3k Privileged Environments" on page 161.

3.1 Introduction

Hardware breakpoints compare addresses and data of executed instructions, including data load/store accesses. Instruction breakpoints can be set even on addresses in ROM areas, and data breakpoints can cause debug exceptions on specific data accesses. Instruction and data hardware breakpoints are alike in many aspects, and are described in parallel in the following sections. When the term "breakpoint" is used in this chapter, then the reference is to a "hardware breakpoint", unless otherwise explicitly noted.

The breakpoints provide the following key features:

- From zero to 15 instruction breakpoints can be implemented to cause debug exceptions on executed instructions, both in ROM and RAM. Bit masking is provided for virtual address compares, and masking of compares with ASID (optional) is also provided.
- From zero to 15 data breakpoints can be implemented to cause debug exceptions on data accesses. Bit masking is provided for virtual address compares, masking of compares with ASID (optional) is provided, optional data value compares allows masking at byte level, and qualification on byte access and access type is possible.
- Registers for setup and control are memory mapped in the drseg segment, accessible in Debug Mode only.

- Breakpoints have several implementation options to ease integration with various microarchitectures.

Hardware breakpoints require the implementation of the Debug Control Register (DCR).

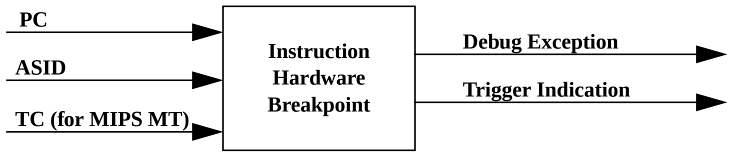
Several additional options are possible for breakpoints, as described in the following subsections.

For EJTAG features, there are no difference between a reset and a soft reset occurring to the processor; they behave identically in both Debug Mode and Non-Debug Mode. References to reset in the following therefore refers to both reset (hard reset) and soft reset.

3.1.1 Instruction Breakpoint Features

Figure 3.2 shows an overview of the instruction breakpoint feature. The feature compares the virtual address (PC) and the ASID of the executed instructions with each instruction breakpoint, applying masking on address and ASID. When an enabled instruction breakpoint matches the PC and ASID, a debug exception and/or a trigger is generated, and an internal bit in an instruction breakpoint register is set to indicate that a match occurred. If the processor implements the MIPS MT ASE, then a match for the TC (Thread Context Id) may also be enabled and required.

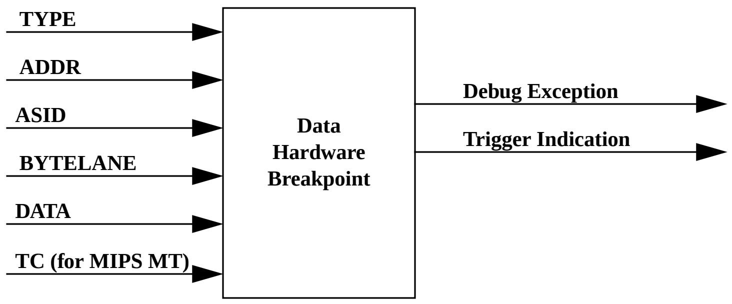
Figure 3.1 Instruction Breakpoint Overview



3.1.2 Data Breakpoint Features

Figure 3.2 shows an overview of the data breakpoint feature. The feature compares the load or store access type (TYPE), the virtual address of the access (ADDR), the ASID, the accessed bytes (BYTELANE), and data value (DATA) with each data breakpoint, applying masks and/or qualifications on the access properties. If the processor implements the MIPS MT ASE, then a match for the TC (Thread Context Id) may also be enabled and required.

Figure 3.2 Data Breakpoint Overview



When an enabled data breakpoint matches, a debug exception and/or a trigger is generated, and an internal bit in a data breakpoint register is set to indicate that a match occurred. The match is either precise (the debug exception or trigger occurs on the instruction that caused the breakpoint to match) or imprecise (the debug exception or trigger occurs later in the program flow).

3.2 Overview of Instruction and Data Breakpoint Registers

From zero to 15 instruction and data breakpoints can be implemented independently. Implementation of any breakpoint implies that the Debug Control Register (DCR) is implemented.

The InstBrk and DataBrk bits in the DCR register indicate whether there are zero or 1 to 15 implementations of a breakpoint type. If no breakpoints of a specific type are implemented, then none of the registers associated with this breakpoint type are implemented.

If any (1 to 15) breakpoints of a specific type are implemented, then the breakpoint status register associated with that breakpoint type is implemented. The instruction and data break status registers indicate the number of breakpoints for each corresponding type. The number of additional registers depends on the number of implemented breakpoints for the respective breakpoint type.

Registers for ASID compares are only implemented if indicated in the corresponding breakpoint status register.

3.2.1 "Overview of Instruction Breakpoint Registers" and 3.2.2 "Overview of Data Breakpoint Registers" provide overviews of the instruction and data breakpoint registers, respectively. All registers are memory mapped in the drseg segment. All registers are 32 bits wide for 32-bit processors and 64 bits wide for 64-bit processors.

3.2.1 Overview of Instruction Breakpoint Registers

Table 3.1 lists the Instruction Breakpoint registers. The Instruction Breakpoint Status register provides implementation indication and status for instruction breakpoints in general. The 1 to 15 implemented breakpoints are numbered 0 to 14, respectively, for registers and breakpoints. The specific breakpoint number is indicated by "n".

Table 3.1 Instruction Breakpoint Register Summary

Register Mnemonic	Register Name and Description	Reference	Compliance Level
IBS	Instruction Breakpoint Status	See Section 3.6.1 on page 47	Required if any instruction breakpoints are implemented, optional otherwise.
IBAn	Instruction Breakpoint Address n	See Section 3.6.2 on page 48	Required with instruction breakpoint n, optional otherwise.
IBMn	Instruction Breakpoint Address Mask n	See Section 3.6.3 on page 48	
IBASIDn	Instruction Breakpoint ASID n	See Section 3.6.4 on page 49	Required with instruction breakpoint n, optional otherwise. Not implemented if ASIDsup bit in IBS is 0 (zero).
IBCn	Instruction Breakpoint Control n	See Section 3.6.5 on page 50	Required with instruction breakpoint n, optional otherwise.

Register addresses are shown in Section 3.6 on page 46.

3.2.2 Overview of Data Breakpoint Registers

Table 3.2 lists the Data Breakpoint Registers. The Data Breakpoint Status register provides implementation indication and status for data breakpoints in general. The 1 to 15 implemented breakpoints are numbered 0 to 14, respectively, for registers and breakpoints. The specific breakpoint number is indicated by "n". The registers for data value compares are only implemented if the value compares for the data breakpoints are implemented, which occurs when either the NoLVmatch bit or the NoSVmatch bit in the DBS is 0.

Table 3.2 Data Breakpoint Register Summary

Register Mnemonic	Register Name and Description	Reference	Compliance Level
DBS	Data Breakpoint Status	See Section 3.7.1 on page 52	Required if any data breakpoints are implemented, optional otherwise.
DBAn	Data Breakpoint Address n	See Section 3.7.2 on page 53	Required with data breakpoint n, optional otherwise.
DBMn	Data Breakpoint Address Mask n	See Section 3.7.3 on page 54	
DBASIDn	Data Breakpoint ASID n	See Section 3.7.4 on page 54	Required with data breakpoint n, optional otherwise. Not implemented if ASIDsup bit in DBS is 0 (zero).
DBCn	Data Breakpoint Control n	See Section 3.7.5 on page 55	Required with data breakpoint n, optional otherwise.
DBVn	Data Breakpoint Value n	See Section 3.7.6 on page 58	Required with data breakpoint n, optional otherwise. Only implemented with value compares, shown in DBS.

Register addresses are shown in Section 3.7 on page 51.

3.3 Conditions for Matching Breakpoints

A number of conditions must be fulfilled in order for a breakpoint to match on an executed instruction or a data access. These conditions are described in the following subsections. A breakpoint only matches for instructions executed in Non-Debug Mode, never due to instructions executed in Debug Mode.

The match of an enabled breakpoint generates a debug exception as described in Section 3.4 on page 43 and/or a trigger indication as described in Section 3.5 on page 45. The BE and/or TE bits in the IBCn or DBCn registers enable the breakpoints for breaks and triggers, respectively.

It is implementation dependent whether or not a breakpoint stalls the processor in order to evaluate the match expression; for example, if required for timing reasons or in order to wait on a scheduled load to return for evaluation of a data breakpoint with a data value compare. In some cases, stalling is avoided with imprecise data breakpoints, as described in Section 3.4.2 on page 43.

3.3.1 Conditions for Matching Instruction Breakpoints

When an instruction breakpoint is enabled, that breakpoint is evaluated in Non-Debug Mode with the instruction boundary address (the lowest address of a byte in the instruction) of every executed instruction. The instruction breakpoint is also evaluated on addresses usually causing an Address Error exception, a TLB exception, or other exceptions. It is thereby possible to cause a Debug Instruction Break exception on the destination address of a jump, even if a jump to that address would cause an Address Error exception. The breakpoint is not evaluated on instructions from speculative fetches or execution.

A match of an instruction breakpoint depends on a number of parameters, shown in Table 3.3. The fields in the instruction breakpoint registers are in the form $\text{REG}_{\text{FIELD}}$.

Table 3.3 Instruction Breakpoint Condition Parameters

Parameter	Description	Width
ASID	ASID field in EntryHi CP0 register.	8 bits
$\text{IBCn}_{\text{ASIDuse}}$	Use ASID value in compare for instruction breakpoint *n*: **Encoding** | **Meaning** 0 | Do not use ASID value in compare 1 | Use ASID value in compare	1 bit
$\text{IBASIDn}_{\text{ASID}}$	Conditional Instruction breakpoint n ASID value for comparing.	8 bits
PC	Virtual address of instruction boundary or target for jump/branch.	32 / 64 bits
ISAmode	Used only when MIPS16e ISA support is implemented. It indicates the ISA mode for the executed instruction or the mode at the target of a jump/branch: **Encoding** | **Meaning** 0 | 32-bit MIPS instruction 1 | MIPS16e instruction	1 bit
IBAn_{IBA}	Instruction breakpoint n address for compare with conditions.	32 / 64 bits
IBMn_{IBM}	Instruction breakpoint n address mask condition: **Encoding** | **Meaning** 0 | Corresponding address bit compared 1 | Corresponding address bit masked	32 / 64 bits
$\text{IBCn}_{\text{TCuse}}$	Thread Context (TC) value used in compare for instruction breakpoint *n*: **Encoding** | **Meaning** 0 | Do not use TC value in compare 1 | Use TC value in compare	1 bit
IBCn_{TC}	TC id value	8 bits max

The PC, IBAn_{IBA}, and IBMn_{IBM} fields are 32 bits wide for 32-bit processors and 64 bits wide for 64-bit processors.

The equation that determines the match is shown below with "C"-like operators. In the equation, 0 means all bits are 0's, and ~0 means all bits are 1's. The widths are similar to the widths of the parameters. The match equation is IB_match, and is dependent on whether MIPS16e is supported or not.

If there is no support for MIPS16e then the IB_match equation is:

```
IB_match =
(!IBCn_TCuse || ( TC == IBCn_TC ) ) &&
( ! IBCn_ASIDuse || ( ASID == IBASIDn_ASID ) ) &&
( ( IBMn_IBM | ~ ( PC ^ IBAn_IBA ) ) == ~0 )
```

If MIPS16e is supported then the IB_match equation is shown below, in which case the ISAmode bit is compared with bit 0 of IBAn_{IBA} instead of a compare with bit 0 in PC:

```
IB_match =
(!IBCn_TCuse || ( TC == IBCn_TC ) ) &&
( ! IBCn_ASIDuse || ( ASID == IBASIDn_ASID ) ) &&
( ( IBMn_IBM | ~ ( ( ( PC[MSB:1] << 1 ) + ISAmode ) ^ IBAn_IBA ) ) == ~0 )
```

The IB_match equation also applies to 64-bit processors running in 32-bit addressing mode, in which case all 64 bits are compared between the PC and the IBAn_{IBA} register.

The match indication for instruction breakpoints is always precise; that is, it is indicated on the instruction causing the IB_match to be true.

It is implementation dependent for an instruction breakpoint to match when the memory system does not ever respond to the fetch or generates a bus error from a system watchdog. If no match occurs, then the processor hangs without the instruction breakpoint generating either a debug exception or a trigger.

3.3.2 Conditions for Matching Data Breakpoints

When a data breakpoint is enabled, that breakpoint is evaluated in Non-Debug Mode with both the access address of every data access due to load/store instructions (including loads/stores of coprocessor registers) and the address causing address errors upon data access. Data breakpoints are not evaluated with addresses from PREF (prefetch) or CACHE instructions. It is implementation dependent whether an SC or SCD instruction causes a data breakpoint if all conditions would cause a match, but the SC or SCD instruction would fail because the LLbit is 0.

The concept “data bus” is used in the following to denote the bytes accessed and the data value transferred in a load/store operation. In this notation data bus refers to the naturally-aligned memory word (for 32-bit processors) or doubleword (for 64-bit processors) containing the accessed address referred to as ADDR. This notation is independent of the actual width of the processor bus, e.g., the “data bus” width of a 64-bit processor is 64, even if that processor has a 32-bit processor bus.

A match of the data breakpoint depends on a number of parameters, shown in Table 3.4. The fields in the data breakpoint registers are in the form $\text{REG}_{\text{FIELD}}$.

Table 3.4 Data Breakpoint Condition Parameters

Reference	Description	Width
TYPE	Data access type is either load or store.	(no width)
$\text{DBCn}_{\text{NoSB}}$	Controls whether condition for data breakpoint is fulfilled on a store access: **Encoding** / **Meaning** 0 — Condition can be fulfilled on store access 1 — Condition is never fulfilled on store access	1 bit
$\text{DBCn}_{\text{NoLB}}$	Controls whether condition for data breakpoint is fulfilled on a load access: **Encoding** / **Meaning** 0 — Condition can be fulfilled on load access 1 — Condition is never fulfilled on load access	1 bit
ASID	ASID field in EntryHi CP0 register.	8 bits

Table 3.4 Data Breakpoint Condition Parameters (Continued)

Reference	Description	Width
$DBCn_{ASIDuse}$	ASID value used in compare for data breakpoint *n*: **Encoding** – **Meaning**; 0 – Do not use ASID value in compare; 1 – Use ASID value in compare	1 bit
$DBASIDn_{ASID}$	Conditional Data breakpoint n ASID value for comparison.	8 bits
ADDR	With one exception, virtual address of data access, effective address. The exception is the LUXC1 and SUXC1 instructions in which the lower three bits of the effective address are ignored (forced to zero for the operation). In this case, ADDR is the effective address with bits 2:0 forced to zero.	32 / 64 bits
$DBAn_{DBA}$	Data breakpoint n address for compare with conditions.	32 / 64 bits
$DBMn_{DBM}$	Conditional Data breakpoint n address mask: **Encoding** – **Meaning**; 0 – Corresponding address bit compared; 1 – Corresponding address bit masked	32 / 64 bits
BYTELANE	Byte lane access indication, where BYTELANE[0] is 1 only if the byte at bits [7:0] of the data bus is accessed, BYTELANE[1] is 1 only if the byte at bits [15:8] of the data bus is accessed, etc.	4 / 8 bits
$DBCn_{BAI}$	Determines whether access is ignored to specific bytes. BAI[0] controls ignore of access to the byte at bits [7:0] of the data bus, BAI[1] ignores access to byte at bits [15:8] of the data bus, etc.: **Encoding** – **Meaning**; 0 – Condition depends on access to corresponding byte; 1 – Access for corresponding byte is ignored	4 / 8 bits
DATA	Data value from the data bus.	32 / 64 bits
$DBVn_{DBV}$	Conditional Data breakpoint n data value for compare.	32 / 64 bits
$DBCn_{BLM}$	Conditional Byte lane mask for value compare on data breakpoint. BLM[0] masks byte at bits [7:0] of the data bus, BLM[1] masks byte at bits [15:8], etc.: **Encoding** – **Meaning**; 0 – Compare corresponding byte lane; 1 – Mask corresponding byte lane	4 / 8 bits
$DBCn_{TCuse}$	Thread Context (TC) value used in compare for data breakpoint *n*: **Encoding** – **Meaning**; 0 – Do not use TC value in compare; 1 – Use TC value in compare	1 bit
$DBCn_{TC}$	TC id value	8 bits max
$DBCn_{IVM}$	Indicates whether or not to invert the data value match	1 bit

The ADDR, DBAn$_{DBA}$, DBMn$_{DBM}$, DATA, and DBVn$_{DBV}$ fields are 32 bits wide for 32-bit processors and 64 bits wide for 64-bit processors. The BYTELANE, DBCn$_{BLM}$, and DBCn$_{BAI}$ fields are four bits wide for 32-bit processors and eight bits wide for 64-bit processors. The width is indicated as "N" in the equations below.

The match equations are shown below with "C"-like operators. In the equation, 0 means all bits are 0's, and ~0 means all bits are 1's. The bit widths are similar to the widths of the parameters.

DB_match is the overall match equation (the DB_addr_match, DB_no_value_compare, and DB_value_match equations in the DB_match equation are defined below):

```
DB_match =
(!DBCn$_{TCuse}$ || ( TC == DBCn$_{TC}$ ) ) &&
( ( ( TYPE == load ) && ! DBCn$_{NoLB}$ ) || ( ( TYPE == store ) && ! DBCn$_{NoSB}$ ) ) &&
DB_addr_match && ( DB_no_value_compare || DB_value_match )
```

DB_addr_match is defined as:

```
DB_addr_match =
( ! DBCn$_{ASIDuse}$ || ( ASID == DBASIDn$_{ASID}$ ) ) &&
( ( DBMn$_{DBM}$ | ~ ( ADDR ^ DBAn$_{DBA}$ ) ) == ~0 ) &&
( ( ~ DBCn$_{BAI}$ & BYTELANE ) != 0 )
```

The DB_addr_match equation also applies to 64-bit processors running in 32-bit addressing mode, in which case all 64 bits are compared between the ADDR and the DBAn$_{DBA}$ field. Please note the special case used for ADDR for the LUXC1 and SUXC1 instructions as described in Table 3.4.

DB_no_value_compare is defined as:

```
DB_no_value_compare =
( ( DBCn$_{BLM}$ | DBCn$_{BAI}$ | ~ BYTELANE ) == ~0 )
```

If a data value compare is indicated on a breakpoint, then DB_no_value_compare is false, and if there is no data value compare then DB_no_value_compare is true. Note that a data value compare is a run-time property of the breakpoint if (DBCn$_{BLM}$ | DBCn$_{BAI}$) is not ~0, because DB_no_value_compare then depends on BYTELANE provided by the load/store instructions. The DBC$_{IVM}$ bit inverts the sense of the match. If set, the value match term will be high if the data value is not the same as the data in the DBVn register.

If a data value compare is required, then the data value from the data bus is compared and masked with the registers for the data breakpoint, as shown in the DB_value_match equation:

```
DB_value_match =
DBCn$_{IVM}$ ^
( ( DATA[7:0] == DBVn$_{DBV[7:0]}$ ) || ! BYTELANE[0] || DBCn$_{BLM[0]}$ || DBCn$_{BAI[0]}$ ) &&
( ( DATA[15:8] == DBVn$_{DBV[15:8]}$ ) || ! BYTELANE[1] || DBCn$_{BLM[1]}$ || DBCn$_{BAI[1]}$ ) &&
......
( ( DATA[8*N-1:8*N-8] == DBVn$_{DBV[8*N-1:8*N-8]}$ ) ||
    ! BYTELANE[N-1] || DBCn$_{BLM[N-1]}$ || DBCn$_{BAI[N-1]}$ )
```

Data breakpoints depend on endianess, because values on the byte lanes are used in the equations. Thus it is required that the debug software programs the breakpoints accordingly to endianess.

It is implementation dependent for a data breakpoint to match when the memory system does not ever respond to the data access or generates a bus error from a system watchdog. If no match occurs, then the processor hangs without the data breakpoint generating a debug exception or trigger.

3.3.2.1 Inverting the Data Value Match Condition

EJTAG specification 4.00 and above introduces the concept of inverting the sense of the data value match. This is an optional feature whose presence is indicated by bit 15 in the Debug Control Register (DCR_{IVM}). When present, bit 1 in the Data Break Control register DBC_{IVM} indicates whether the match sense should be inverted during execution.

3.3.2.2 Data Breakpoints in case of Unaligned Address

Unaligned addresses can result from explicit halfword, word, and doubleword accesses (for example, if an effective address of 0x01 is used as source of a Load Halfword (LH) instruction). The ADDR used in the comparison is the effective address. The BYTELANE value is defined according to Table 3.5 for a 32-bit processor and to Table 3.6 for a 64-bit processor.

Table 3.5 BYTELANE at Unaligned Address for 32-bit Processors

	ADDR			**BYTELANE[3:0]**	
Size	**[2]**	**[1]**	**[0]**	**Little Endian**	**Big Endian**
Halfword	x	0	x	0011_2	1100_2
	x	1	x	1100_2	0011_2
Word	x	x	x	1111_2	
‘x’ denotes don’t care					

Table 3.6 BYTELANE at Unaligned Address for 64-bit Processors

	ADDR			**BYTELANE[7:0]**	
Size	**[2]**	**[1]**	**[0]**	**Little Endian**	**Big Endian**
Halfword	0	0	x	00000011_2	11000000_2
	0	1	x	00001100_2	00110000_2
	1	0	x	00110000_2	00001100_2
	1	1	x	11000000_2	00000011_2
Word	0	x	x	00001111_2	11110000_2
	1	x	x	11110000_2	00001111_2
Doubleword	x	x	x	11111111_2	
‘x’ denotes don’t care					

With the above well-defined values of BYTELANE, the behavior is well-defined for data breakpoints without value compares on operations with unaligned addresses. The BLM field in the DBCn register can be used to avoid value compares if all BLM bits are set to 1.

If the data breakpoint depends on a value compare, then loads will cause an Address Error exception, and for stores the data value (DATA) is UNPREDICTABLE. This UNPREDICTABLE data can cause match of a data breakpoint on a store, but an implementation can choose never to indicate a match on data breakpoints depending on value compare if having unaligned address.

If a debug exception is taken on the store then the debug handler can investigate the processor state and thereby determine if the address was unaligned and UNPREDICTABLE store data for the memory access thereby caused the debug exception. If a debug exception is not taken for the store, then an Address Error exception is taken. So, in both cases it is possible for debug software to detect the bug. The BLM field in the DBCn register can be used to avoid compare on UNPREDICTABLE data, in case all of the BLM bits are set to 1.

If the data breakpoint is used as a triggerpoint (see Section 3.5 on page 45) then a BS bit might be set after a compare with UNPREDICTABLE data; however, an Address Error exception occurs in this case thereby making it possible to detect the bug.

3.3.2.3 Match for Data Breakpoint with Value Compare on Bus or Cache Error

If a data value compare is required to evaluate a data breakpoint, the DB_no_value_compare equation is false (see Section 3.3.2 on page 38). However, if a bus or cache error occurs on the load, then there is no valid data to use in the compare. This case has two possibilities:

- The match will fail.
- The match will compare on invalid data, and then indicate a pending bus or cache error through the DBusEP or CacheEP bits in the Debug register, if a debug exception is taken. This occurrence might cause a trigger indication to be set on the compare with invalid data.

A bus or cache error on a store does not affect the data breakpoint compare.

Refer to Section 3.8.3 on page 58 for recommendations on implementing data breakpoint compares on invalid data.

3.3.2.4 Precise Match for Data Breakpoints

A precise match for a data breakpoint occurs when the match equation can be fully evaluated at the time the load/store instruction is executed. A true DB_match can thereby be indicated on the very same instruction causing the DB_match equation to be true.

Matches on data breakpoints without data value compares are always precise. Accesses using data value compares are either imprecise or precise depending on the implementation and specific access.

3.3.2.5 Imprecise Match for Data Breakpoints

An imprecise match for a data breakpoint occurs when the match equation cannot be fully evaluated at the time the load/store instruction is executed. This case occurs when the processor is not stalled on a scheduled load and a data breakpoint must compare on the data value returned by the load. If the breakpoint matches, then the DB_match equation is true later in the execution flow rather than at the same time as load/store instruction that caused the load/store access to match.

Only data breakpoints with value compares can be imprecise, in which case the breakpoints can be imprecise for all or some of those accesses depending on the implementation.

3.3.3 Precise Exceptions on Data Value Match Breaks

When the EJTAG hardware implements data value match breaks to be taken precisely, the core EJTAG hardware on obtaining the data value will match the value and cause an exception to be taken on the load instruction. In this situation, the data value is already read out from its source location and brought to the processor. When the exception handler has taken the exception, the DEPC points to the load instruction (because the exception is taken precisely), and the load instruction reexecutes on a return from exception. If the load value was being read from regular memory, then this is usually not an issue. But in a situation where the load data was coming from a special FIFO or I/O register, this instruction can not be reexecuted without altering the state of the peripheral or special memory. To handle this type of situation, when the EJTAG hardware implements precise data value exceptions, is also expected to keep the load data value in a drseg register. This allows the debug exception handler to re-execute this instruction in software using this data value. The debug handler must also re-calculate the new DEPC value and update it before executing the DERET instruction. This Load Data Value register is at drseg address 0x2FF0.

This is an optional feature of regular EJTAG introduced in revision 4.00 and above and the presence of this feature is indicated by bit 14 (DVM) of the DCR register.

3.4 Debug Exceptions from Breakpoints

This section describes how to set up instruction and data breakpoints to generate debug exceptions when the match conditions are true.

3.4.1 Debug Exception Caused by Instruction Breakpoint

The BE bit in the IBCn register must be set for an instruction breakpoint to be enabled. A Debug Instruction Break exception occurs when the IB_match equation is true (see Section 3.3.1 on page 36). The corresponding BS bit in the IBS register is set when the breakpoint generates the debug exception. Note that the BE bit alone enables the break point exception, irrespective of whether or not the TE bit is set (see Section 3.5 on page 45).

The Debug Instruction Break exception is precise, so the DEPC register and DBD bit in the Debug register (see Section 6.7 on page 104) point to the instruction that caused the IB_match equation to be true.

The instruction receiving the debug exception only updates the debug related registers. That instruction will not cause any loads/stores to occur. Thus a debug exception from a data breakpoint cannot occur at the same time an instruction receives a Debug Instruction Break exception.

The debug handler usually returns to the instruction causing the Debug Instruction Break exception, whereby the instruction is executed. Debug software must disable the breakpoint when returning to the instruction, otherwise the Debug Instruction Break exception will reoccur. An alternative is for debug software to emulate the instruction(s) in software and change the DEPC accordingly.

3.4.2 Debug Exception by Data Breakpoint

The BE bit in the DBCn register must be set for a data breakpoint to be enabled. A debug exception occurs when the DB_match condition is true (see Section 3.3.2 on page 38). A matching data breakpoint generates either a precise or an imprecise debug exception. Note that the BE bit alone enables the break point exception, irrespective of whether or not the TE bit is set (see Section 3.5 "Breakpoints Used as Triggerpoints").

Refer to Section 3.8.4 on page 59 for additional information on precise and imprecise debug exceptions.

3.4.2.1 Debug Data Break Load/Store Exception as a Precise Debug Exception

A Debug Data Break Load/Store exception occurs when a data breakpoint indicates a precise match. In this case, the DEPC register and DBD bit in the Debug register point to the load/store instruction that caused the DB_match equation (see Section 3.3.2 on page 38) to be true, and the corresponding BS bit in the DBS register is set. Details about behavior of the instruction causing the debug exception is shown in Table 3.7.

Table 3.7 Behavior on Precise Exceptions from Data Breakpoints

Instruction and Data Breakpoint	Load/Store Instruction Execution	Destination Register	External Memory System Access
Store wo/w value match	Not completed	Not updated[1]	Store to memory is not committed
Load without value match		Not updated[2]	Load from memory does not occur
Load with value match			Load from memory does occur

1. This applies to the Store Conditional Word/Doubleword (SC/SCD) instructions
2. This includes side effects like for the Load Linked Word/Doubleword (LL/LLD) instructions

Thus in the case a data breakpoint with data value compare is set up on a load instruction, then the load does occur from the external memory, since the data value is required to evaluate the match condition, but the destination register is not updated, so the loaded value is simply discarded.

The rules shown in Table 3.8 describe update of the BS bits when several data breakpoints match the same access and generate a debug exception.

Table 3.8 Rules for Update of BS Bits on Precise Exceptions from Data Breakpoints

	Breakpoints That Matches...		Update of BS Bits for Matching Data Breakpoints	
Instruction	Without Value Compare	With Value Compare	Without Value Compare	With Value Compare
Load / Store	One or more	None	BS bits set for all	(No matching breakpoints)
Load	One or more	One or more	BS bits set for all	Unchanged BS bits since load of data value does not occur, so match of the breakpoint cant be determined
Load	None	One or more	(No matching breakpoints)	BS bits set for all
Store	One or more	One or more	BS bits set for all	Optional to either set BS bits for all, or change none of the BS bits
Store	None	One or more	(No matching breakpoints)	BS bits set for all

Any BS bit set prior to the match and debug exception is kept set, since only debug software can clear the BS bits.

The debug handler usually returns to the instruction that caused the Debug Data Break Load/Store exception, whereby the instruction is re-executed. This re-execution results in a repeated load from system memory after a data breakpoint with a data value compare on a load, because the load occurred previously in order to allow evaluation of the breakpoint as described above. Memory-mapped devices with side effects on loads must allow such reloads, or debug software should alternatively avoid setting data breakpoints with data value compares on the address of such devices. Debug software must disable breakpoints when returning to the instruction, otherwise the Debug Data Break Load/Store exception will reoccur. An alternative is for debug software to emulate the instruction in software and change the DEPC accordingly.

3.4.2.2 Debug Data Break Load/Store Exception as an Imprecise Debug Exception

A Debug Data Break Load/Store Imprecise exception occurs when a data breakpoint indicates an imprecise match. In this case, the DEPC register and DBD bit in the Debug register point to an instruction later in the execution flow rather than at the load/store instruction that caused the DB_match equation to be true.

The load/store instruction causing the Debug Data Break Load/Store Imprecise exception always updates the destination register and completes the access to the external memory system. Therefore this load/store instruction is not re-executed on return from the debug handler, because the DEPC register and DBD bit do not point to that instruction.

Several imprecise data breakpoints can be pending at a given time, if the bus system supports multiple outstanding data accesses. The breakpoints are evaluated as the accesses finalize, and a Debug Data Break Load/Store Imprecise exception is generated only for the first one matching. Both the first and succeeding matches cause corresponding BS bits and DDBLImpr/DDBSImpr to be set, but no debug exception is generated for succeeding matches because the processor is already in Debug Mode. Similarly, if a debug exception had already occurred at the time of the first match (for example, due to a precise debug exception), then all matches cause the corresponding BS bits and DDBLImpr/DDBSImpr to be set, but no debug exception is generated because the processor is already in Debug Mode.

The SYNC and EHB instructions, followed by appropriate spacing, (as described in Section 6.2.3.7 on page 88 and Section 6.2.4 on page 89) must be executed before the BS bits and DDBLImpr/DDBSImpr bits are respectively accessed for read or write. This delay ensures that these bits are fully updated.

Any BS bit set prior to the match and debug exception are kept set, because only debug software can clear the BS bits.

3.5 Breakpoints Used as Triggerpoints

Software can set up both instruction and data breakpoints such that a matching breakpoint does not generate a debug exception, but sends an indication through the BS bit only. But note that if the BE bit is set, then a debug exception will be generated, even if the TE bit is set. The TE bit in the IBCn or DBCn register controls whether an instruction or data breakpoint, respectively, is used as a triggerpoint. Triggerpoints are evaluated for matches under the same criteria as breakpoints.

The BS bit in the IBS or DBS register is set for a triggerpoint when the respective IB_match condition (see Section 3.3.1 on page 36) or DB_match condition (see Section 3.3.2 on page 38) is true.

Table 3.9 Actions Resulting from an Instruction/Data Match for Specified BE and TE Bit Values

TE	BE	Breakpoint Exception	BS bit is set in IBS/DBS
0	0	Not taken	No
0	1	Taken	Yes
1	0	Not taken	Yes
1	1	Taken	Yes

For the BS bit to be set for an instruction triggerpoint, either the instruction must be fully executed or an exception must occur on the instruction.

The BS bit for a data triggerpoint can only be set if no exception with higher priority than the Debug Data Break Load/Store exception with address match only occurred on the load/store instruction. For exceptions with equal or lower priority than the Debug Data Break Load/Store exception with address match only, the BS bits are still set for a matching triggerpoint. For example, the BS bit is set even if a TLB or Bus Error exception occurred on the load/store instruction. Data triggerpoints with value compares require the data value to be valid for the BS bit to be set, which is not the case if, for example, a TLB or Bus Error exception occurs on a load instruction. However, for stores, the trigger may compare on UNPREDICTABLE data as described in Section 3.3.2.2 on page 41.

The rules for update of the BS bits are shown in Table 3.10.

Table 3.10 Rules for Update of BS Bits on Data Triggerpoints

Instruction	Without/With Value Compare	BS Bits Update for Triggerpoint
Load / Store	Without value compare	BS bit set if no exception with higher priority than the Debug Data Break Load/Store exception, with address match only, occurred on the instruction.
Load	With value compare	BS bit set if no exception with higher priority than the Debug Data Break Load exception, with address and data value match, occurred on the instruction.
Store	With value compare	BS bit is set if no exception occurred on the instruction, and is optional to be if an exception with equal or lower priority than the Debug Data Break Store exception, with address match only, occurred on the instruction, with the requirement that either all the relevant BS bits are set, or none are changed.

Data breakpoints with imprecise matches generate imprecise triggers when enabled by the TE bit.

Note that trigger indications by BS may be set based on compare with UNPREDICTABLE data, as described in (see Section 3.3.2.2 on page 41).

A triggerpoint match can be indicated on an optional internal signal or chip pin.

3.6 Instruction Breakpoint Registers

This section describes the instruction breakpoint registers for MIPS32 and MIPS64 processors, and other R4k privileged environment implementations of 32-bit and 64-bit processors. These registers provide status and control for the instruction breakpoints. All registers are in the drseg segment. The 1 to 15 implemented breakpoints are numbered 0 to 14, respectively, for registers and breakpoints. The specific breakpoint number is indicated by "n" in the range 0 to 15 depending on the implemented number of instruction breakpoints. The registers and their respective addresses offsets are shown in Table 3.11. For a description of the two registers IBCC and IBPC used for complex breakpoints, see Section 4.3.2 on page 66 and Section 4.3.4 on page 67 respectively.

Table 3.11 Instruction Breakpoint Register Mapping

Offset in drseg	Register Mnemonic	Register Name and Description
0x1000	IBS	Instruction Breakpoint Status
0x1100 + 0x100 * n	IBAn	Instruction Breakpoint Address n
0x1108 + 0x100 * n	IBMn	Instruction Breakpoint Address Mask n
0x1110 + 0x100 * n	IBASIDn	Instruction Breakpoint ASID n
0x1118 + 0x100 * n	IBCn	Instruction Breakpoint Control n
0x1120 + 0x100 * n	IBCCn	Instruction Breakpoint Complex Control n
0x1128 + 0x100 * n	IBPCn	Instruction Breakpoint Pass Counter n

3.6.1 Instruction Breakpoint Status (IBS) Register

Compliance Level: Required if any instruction breakpoints are implemented, optional otherwise.

The Instruction Breakpoint Status (IBS) register holds implementation and status information about the instruction breakpoints. It is located at drseg segment offset 0x1000. The ASIDsup bit applies to all instruction breakpoints.

Figure 3.3 shows the format of the IBS register; Table 3.12 describes the IBS register fields.

Figure 3.3 IBS Register Format

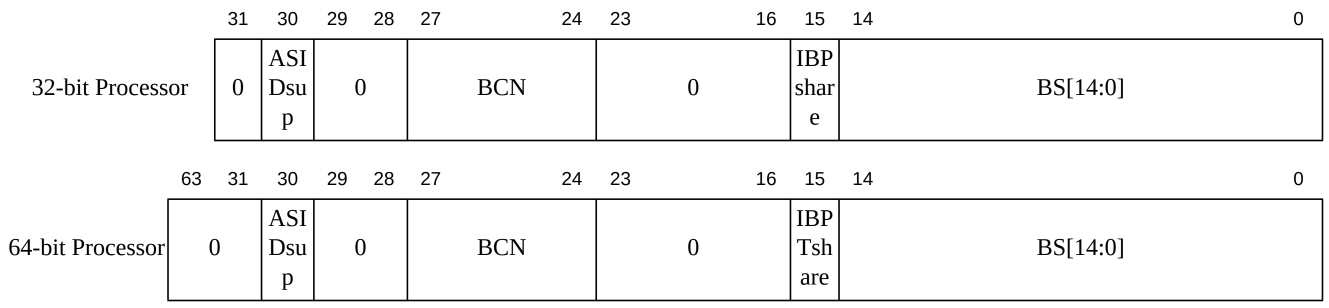


Table 3.12 IBS Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
ASIDsup	30	Indicates if ASID compare is supported in instruction breakpoints: Encoding / Meaning: 0 — No ASID compare 1 — ASID compare (IBASIDn register implemented) ASID support indication does not guarantee a TLB-type MMU, because the same breakpoint implementation can be used with processors having all different types of MMUs.	R	Preset	Required
BCN	27:24	Number of instruction breakpoints implemented: Encoding / Meaning: 0 — Reserved 1-15 — Number of instructions breakpoints	R	Preset	Required
IBPshare	15	Determines whether the Instruction breakpoints are shared across the different VPEs of the processor, or are implemented per-VPE. Encoding / Meaning: 0 — Not shared 1 — Shared across VPEs	R	Preset	Required in MIPS MT is implemented. Otherwise Reserved.

Table 3.12 IBS Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
BS[14:0]	14:0	Break Status (BS) bit for breakpoint n is at BS[n], where n is 0 to 14. A bit is set to 1 when the condition for its corresponding breakpoint has matched. The number of BS bits implemented corresponds to the number of breakpoints indicated by the BCN field. Debug software is expected to clear the bits before use, because reset does not clear these bits. Bits not implemented are read-only (R) and read as zeros.	R/W0	Undefined	Required for bits at implemented breakpoints, other bits not implemented
0	MSB:31, 29:28, 23:16	Must be written as zeros; return zeros on read.	0	0	Reserved

3.6.2 Instruction Breakpoint Address n (IBAn) Register

Compliance Level: Required with instruction breakpoint n, optional otherwise.

The Instruction Breakpoint Address n (IBAn) register has the virtual address used in the condition for instruction breakpoint n. It is located at drseg segment offset 0x1100 + 0x100 * n.

Figure 3.4 shows the format of the IBAn register; Table 3.13 describes the IBAn register field.

Figure 3.4 IBAn Register Format

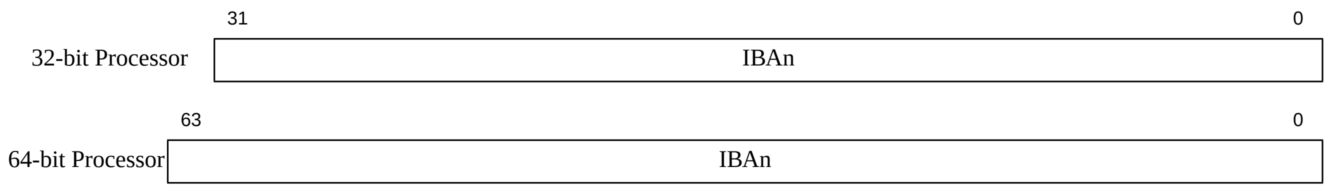


Table 3.13 IBAn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
IBA	MSB:0	Instruction breakpoint virtual address for condition.	R/W	Undefined	Required

3.6.3 Instruction Breakpoint Address Mask n (IBMn) Register

Compliance Level: Required with instruction breakpoint n, optional otherwise.

The Instruction Breakpoint Address Mask n (IBMn) register has the address compare mask used in the condition for instruction breakpoint n. The address that is masked is in the IBAn register. The IBMn register is located at drseg segment offset 0x1108 + 0x100 * n.

Figure 3.5 shows the format of the IBMn register; Table 3.14 describes the IBMn register field.

Figure 3.5 IBMn Register Format

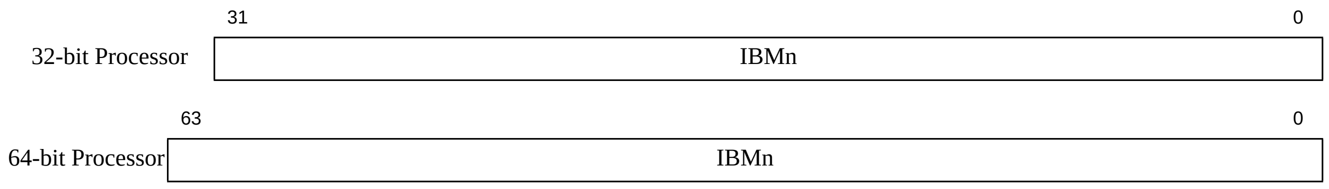


Table 3.14 IBMn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
IBM	MSB:0	Instruction breakpoint address mask for condition:	R/W	Undefined	Required

Encoding	Meaning
0	Corresponding address bit compared
1	Corresponding address bit masked

3.6.4 Instruction Breakpoint ASID n (IBASIDn) Register

Compliance Level: Required with instruction breakpoint n if the ASIDsup bit in the IBS register is 1, optional otherwise.

The Instruction Breakpoint ASID n (IBASIDn) register has the ASID value used in the compare for instruction breakpoint n. It is located at drseg segment offset 0x1110 + 0x100 * n.

Figure 3.6 shows the format of the IBASIDn register; Table 3.15 describes the IBASIDn register fields. The width of the ASID field for the compare is 8 bits. It is identical to the width of the ASID field in the EntryHi register used with the TLB-type MMU.

Figure 3.6 IBASIDn Register Format

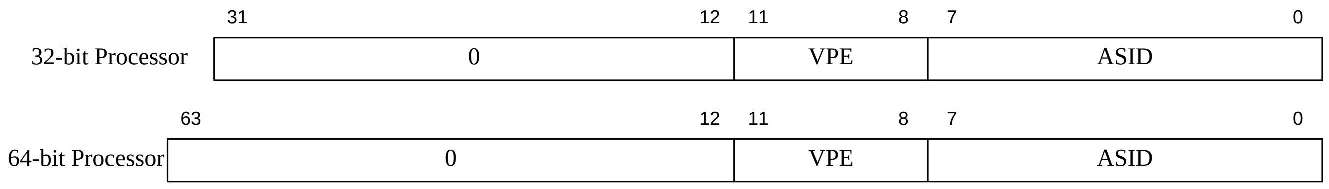


Table 3.15 IBASIDn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
ASID	7:0	Instruction breakpoint ASID value for compare.	R/W	Undefined	Required
VPE	11:8	This field indicates the value of the VPE id to use for comparison and is used only if VPEuse in IBCn register is 1 and the breakpoints are shared across VPEs. If the breakpoints are not shared, then these bits read zero, and writes are ignored.	R/W	Undefined	Required if MIPS MT is implemented. Otherwise Reserved.
0	MSB:12	Must be written as zeros; return zeros on read.	0	0	Reserved

3.6.5 Instruction Breakpoint Control n (IBCn) Register

Compliance Level: Required with instruction breakpoint n, optional otherwise.

The Instruction Breakpoint Control n (IBCn) register determines what constitutes instruction breakpoint n: trigger-point, breakpoint, ASID value inclusion. This register is located at drseg segment offset 0x1118 + 0x100 * n.

Figure 3.7 shows the format of the IBCn register; Table 3.15 describes the IBCn register fields.

Figure 3.7 IBCn Register Format

	31 24	23	22	21 4	3	2	1	0
32-bit Processor	TC	ASID use	TC use	0	VPE use	TE	0	BE

	63 32	31 24	23	22	21 4	3	2	1	0
64-bit Processor	0	TC	ASID use	TC use	0	VPE use	TE	0	BE

Table 3.16 IBCn Register Field Descriptions

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
TC	31:24	The value of TC (thread context) to match in the comparison to determine if the instruction break is to be taken. This comparison is effective only if the TCuse bit is set to 1. Otherwise this TC value is ignored.	R/W	Undefined	Required if MIPS MT is implemented. Otherwise Reserved.
ASIDuse	23	Use ASID value in compare for instruction breakpoint *n*: **Encoding** / **Meaning** 0 — Do not use ASID value in compare 1 — Use ASID value in compare Debug software should only set the ASIDuse if a TLB in the implementation is used by the application software. This bit is read-only and reads as zero, if not implemented.	R/W	Undefined	Required if ASIDsup in IBS register is 1, otherwise not implemented
TCuse	22	Use TC value in comparison for instruction breakpoint n. If TC is not used in the comparison, then the comparison is restricted to the match all TCs in the current VPE if the breakpoints are not shared. If the breakpoints are shared, then they can match all TCs in the processor unless VPEuse is set. **Encoding** / **Meaning** 0 — Do not use TC value in compare 1 — Use TC value in compare	R/W	Undefined	Required if MIPS MT is implemented. Otherwise Reserved.

Table 3.16 IBCn Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
VPEuse	17	Use VPE value in comparison for instruction breakpoint n. This field is used only if the breakpoints are shared across the VPEs of a MT core, that is, the IBPshare bit is set in register IBP. If the breakpoints are not shared, then these bits read zero, and writes are ignored.	R/W	Undefined	Required if MIPS MT is implemented. Otherwise Reserved.
TE	2	Use instruction breakpoint *n* as triggerpoint: **Encoding** / **Meaning** 0: Do not use it as triggerpoint 1: Use it as triggerpoint	R/W	0	Required
BE	0	Use instruction breakpoint *n* as breakpoint: **Encoding** / **Meaning** 0: Do not use it as breakpoint 1: Use it as breakpoint	R/W	0	Required
0	21:4, 1	Must be written as zeros; return zeros on read.	0	0	Reserved

3.7 Data Breakpoint Registers

This section describes the data breakpoint registers for MIPS32 and MIPS64 processors, and other R4k privileged environment implementations of 32-bit and 64-bit processors. These registers provide status and control for the data breakpoints. All registers are in the drseg segment. The 1 to 15 implemented breakpoints are numbered 0 to 14, respectively, for registers and breakpoints. The specific breakpoint number is indicated by “n” in the range 0 to 15 depending on the implemented number of data breakpoints. The registers and their respective addresses offsets are shown in Table 3.17. For a description of the two registers DBCC and DBPC used for complex breakpoints, see Section 4.3.4 on page 67 and Section 4.3.5 on page 69 respectively.

Table 3.17 Data Breakpoint Register Mapping

Offset in drseg	Register Mnemonic	Register Name and Description
0x2000	DBS	Data Breakpoint Status
0x2100 + 0x100 * n	DBAn	Data Breakpoint Address n
0x2108 + 0x100 * n	DBMn	Data Breakpoint Address Mask n
0x2110 + 0x100 * n	DBASIDn	Data Breakpoint ASID n
0x2118 + 0x100 * n	DBCn	Data Breakpoint Control n
0x2120 + 0x100 * n	DBVn	Data Breakpoint Value n
0x2128 + 0x100 * n	DBCCn	Data Breakpoint Complex Control n
0x2130 + 0x100 * n	DBPCn	Data Breakpoint Pass Counter n

3.7.1 Data Breakpoint Status (DBS) Register

Compliance Level: Required if any data breakpoints are implemented, optional otherwise.

The Data Breakpoint Status (DBS) register holds implementation and status information about the data breakpoints. It is located at drseg segment offset 0x2000. The ASIDsup, NoSVmatch, and NoLVmatch fields apply to all data breakpoints.

Figure 3.8 shows the format of the DBS register; Table 3.18 describes the DBS register fields.

Figure 3.8 DBS Register Format

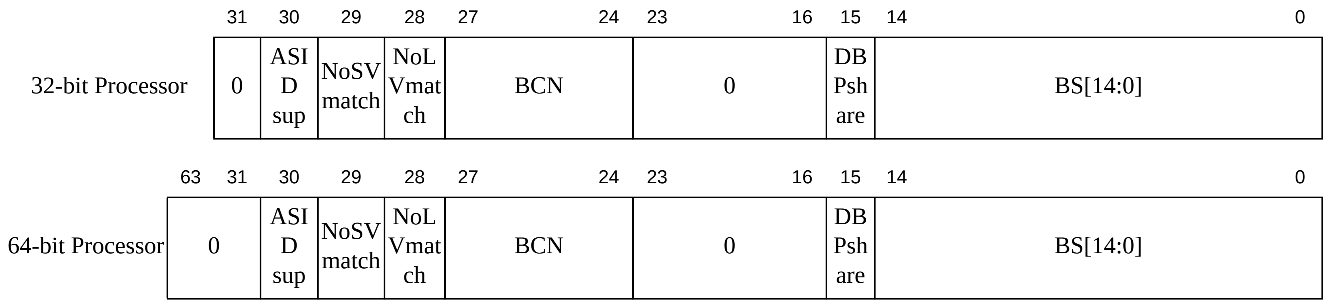


Table 3.18 DBS Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	**Bits**				
ASIDsup	30	Indicates if ASID compare is supported in data breakpoints: **Encoding** / **Meaning** 0 — No ASID compare 1 — ASID compare (DBASIDn register implemented) ASID support indication does not guarantee a TLB-type MMU, because the same breakpoint implementation can be used with processors having all different types of MMUs.	R	Preset	Required
NoSVmatch	29	Indicates if a value compare on a store is supported in data breakpoints: **Encoding** / **Meaning** 0 — Data value and address in condition on store 1 — Address compare only in condition on store	R	Preset	Required

Table 3.18 DBS Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
NoLVmatch	28	Indicates if a value compare on a load is supported in data breakpoints: Encoding / Meaning; 0: Data value and address in condition on load; 1: Address compare only in condition on load	R	Preset	Required
BCN	27:24	Number of data breakpoints implemented: Encoding / Meaning; 0: Reserved; 1-15: Number of data breakpoints	R	Preset	Required
DBPshare	15	Determines whether the Data breakpoints are shared across the different VPEs of the processor, or are implemented per-VPE. Encoding / Meaning; 0: Not shared; 1: Shared across VPEs	R	Preset	Required if MIPS MT is implemented, otherwise Reserved.
BS[14:0]	14:0	Break Status (BS) bit for breakpoint n is at BS[n], where n is 0 to 14. The bit is set to 1 when the condition for its corresponding breakpoint has matched. The number of BS bits implemented corresponds to the number of breakpoints indicated by the BCN bit. Debug software is expected to clear the bits before use, since these are not cleared by reset. Bits not implemented are read-only (R) and read as zeros.	R/W0	Undefined	Required for bits at implemented breakpoints, other bits not implemented
0	MSB:31, 23:16	Must be written as zeros; return zeros on read.	0	0	Reserved

3.7.2 Data Breakpoint Address n (DBAn) Register

Compliance Level: Required with data breakpoint n, optional otherwise.

The Data Breakpoint Address n (DBAn) register has the virtual address used in the condition for data breakpoint n. This register is located at drseg segment offset 0x2100 + 0x100 * n.

Figure 3.9 shows the format of the DBAn register; Table 3.19 describes the DBAn register field.

Figure 3.9 DBAn Register Format

	31 … 0
32-bit Processor	DBAn

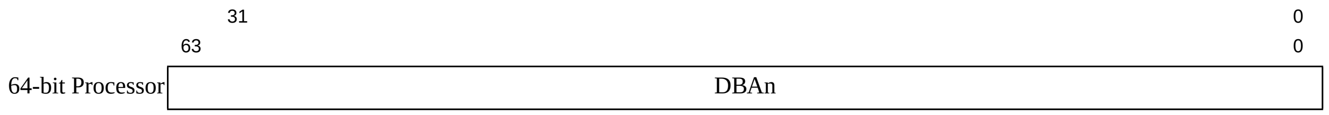


Table 3.19 DBAn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
DBA	MSB:0	Data breakpoint virtual address for condition	R/W	Undefined	Required

3.7.3 Data Breakpoint Address Mask n (DBMn) Register

Compliance Level: Required with data breakpoint n, optional otherwise.

The Data Breakpoint Address Mask n (IBMn) register has the address compare mask used in the condition for data breakpoint n. The address that is masked is in the DBAn register. The DBMn register is located at drseg segment offset 0x2108 + 0x100 * n.

Figure 3.10 shows the format of the DBMn register; Table 3.20 describes the DBMn register field.

Figure 3.10 DBMn Register Format

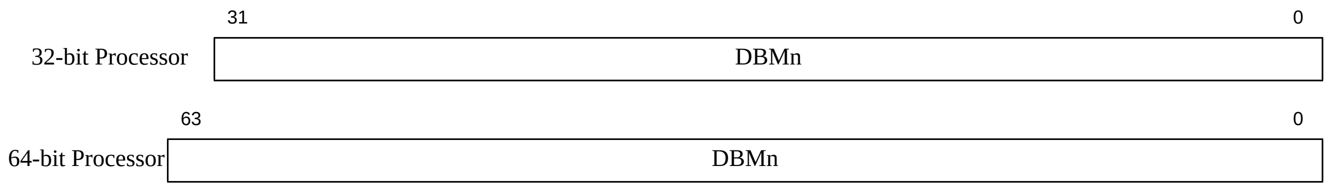


Table 3.20 DBMn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
DBMn	MSB:0	Data breakpoint address mask for condition: **Encoding** / **Meaning** 0 — Corresponding address bit compared 1 — Corresponding address bit masked	R/W	Undefined	Required

3.7.4 Data Breakpoint ASID n (DBASIDn) Register

Compliance Level: Required with data breakpoint n if the ASIDsup bit in the DBS register is 1, optional otherwise.

The Data Breakpoint ASID n (DBASIDn) register has the ASID value used in the compare for data breakpoint n. It is located at drseg segment offset 0x2110 + 0x100 * n.

Figure 3.11 shows the format of the DBASIDn register; Table 3.21 describes the DBASIDn register fields. The width of the ASID field for the compare is 8 bits. It is identical to the width of the ASID field in the EntryHi register used with the TLB-type MMU.

Figure 3.11 DBASIDn Register Format

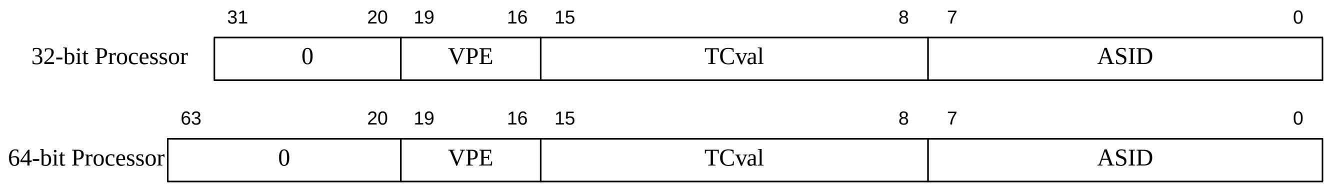


Table 3.21 DBASIDn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	**Bits**				
VPE	11:8	This field indicates the value of the VPE id to use for comparison and is used only if VPEuse in DBCn register is 1 and the breakpoints are shared across VPEs. If the breakpoints are not shared, then these bits read zero, and writes are ignored.	R/W	Undefined	Required if MIPS MT is implemented. Otherwise Reserved.
TCval	15:8	Value of the thread context that caused the Data Breakpoint is jammed into these bits since the data breaks are imprecise. Software can examine these bits to determine which thread context actually caused the data break.	R/W	Undefined	Required if MIPS MT is implemented, otherwise Reserved.
ASID	7:0	Data breakpoint ASID value for compare.	R/W	Undefined	Required
0	MSB:20	Must be written as zeros; return zeros on read.	0	0	Reserved

3.7.5 Data Breakpoint Control n (DBCn) Register

Compliance Level: Required with data breakpoint n, optional otherwise.

The Data Breakpoint Control n (DBCn) register what constitutes data breakpoint n: triggerpoint, breakpoint, ASID value inclusion, load/store access fulfillment, ignore byte access, byte lane mask. This register is located at drseg segment offset 0x2118 + 0x100 * n.

For description of "data bus" notation see Section 3.3.2 on page 38.

Figure 3.12 shows the format of the DBCn register; Table 3.22 describes the DBCn register fields.

Figure 3.12 DBCn Register Format

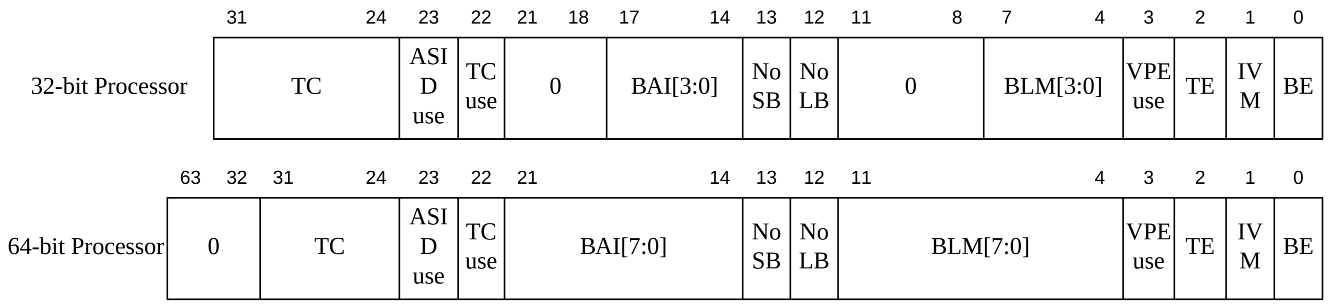


Table 3.22 DBCn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
TC	31:24	The value of TC (thread context) to match in the comparison to determine if the data break is to be taken. This comparison is effective only if the TCuse bit is set to 1. Otherwise this TC value is ignored.	R/W	Undefined	Required in MIPS MT is implemented. Otherwise Reserved.
ASIDuse	23	Use ASID value in compare for data breakpoint *n*: Encoding 0: Do not use ASID value in compare Encoding 1: Use ASID value in compare Debug software should only set the ASIDuse if a TLB in the implementation is used by the application software. This bit is read-only and reads as zero, if not implemented.	R/W	Undefined	Required if ASIDsup in DBS reg. is 1, otherwise not implemented
TCuse	22	Use TC value in comparison for data breakpoint n. Encoding 0: Do not use TC value in compare Encoding 1: Use TC value in compare	R/W	Undefined	Required if ASIDsup in DBS reg. is 1, otherwise not implemented
BAI[:0]	21:14	Byte access ignore. Each bit of this field determines whether a match occurs on an access to a specific byte of the database (BAI[0] controls matching for data bus bits 7:0; BAI[1] controls matching for data bus bits 15:8, etc.)., with the polarity of each bit, as follows: Encoding 0: Condition depends on access to corresponding byte Encoding 1: Access for corresponding byte is ignored A match depends on a reference accessing one or more of the non-ignored bytes. No matches will occur if all bytes are ignored. Debug software must adjust for endianess when programming this field.	R/W	Undefined	Required for byte lanes in implementation, otherwise not implemented
NoSB	13	Controls whether condition for data breakpoint is ever fulfilled on a store access: Encoding 0: Condition can be fulfilled on store access Encoding 1: Condition is never fulfilled on store access	R/W	Undefined	Required

Table 3.22 DBCn Register Field Descriptions (Continued)

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
NoLB	12	Controls whether condition for data breakpoint is ever fulfilled on a load access: **Encoding** **Meaning** 0 Condition can be fulfilled on load access 1 Condition is never fulfilled on load access	R/W	Undefined	Required
BLM[:0]	11:4	Byte lane mask for value compare on data breakpoint. BLM[0] masks byte at bits [7:0] of the data bus, BLM[1] masks byte at bits [15:8], etc.: **Encoding** **Meaning** 0 Compare corresponding byte lane 1 Mask corresponding byte lane Debug software must adjust for endianess when programming this field. BLM[:0] are unimplemented if value compare is not implemented, which is the case when NoSVmatch and NoLVmatch bits in DBS are both 1. Bits are read-only (R) and read as zeros if not implemented.	R/W	Undefined	Required for byte lanes in implementation and if value compare, otherwise not implemented
VPEuse	17	Use VPE value in comparison for instruction breakpoint n. This field is used only if the breakpoints are shared across the VPEs of a MT core, that is, the DBPshare bit is set in register DBP. If the breakpoints are not shared, then these bits read zero, and writes are ignored.	R/W	Undefined	Required if MIPS MT is implemented. Otherwise Reserved.
TE	2	Use data breakpoint n as triggerpoint: **Encoding** **Meaning** 0 Do not use it as triggerpoint 1 Use it as triggerpoint	R/W	0	Required
IVM	1	Used to indicate that the data value match should be inverted.	R/W	Undefined	Required if DCR_{IVM} is 1, otherwise not implemented; Revision 4.00 and above.
BE	0	Use data breakpoint n as breakpoint: **Encoding** **Meaning** 0 Do not use it as breakpoint 1 Use it as breakpoint	R/W	0	Required
0	3	Must be written as zeros; return zeros on read.	0	0	Reserved

3.7.6 Data Breakpoint Value n (DBVn) Register

Compliance Level: Required with data breakpoint n if data value compare is supported (indicated by either NoSVmatch or NoLVmatch bits in DBS being 0), optional otherwise.

The Data Breakpoint Value n (DBVn) register has the value used in the condition for data breakpoint n. It is located at drseg segment offset 0x2120 + 0x100 * n.

Figure 3.13 shows the format of the DBVn register; Table 3.23 describes the DBVn register field.

Figure 3.13 DBVn Register Format

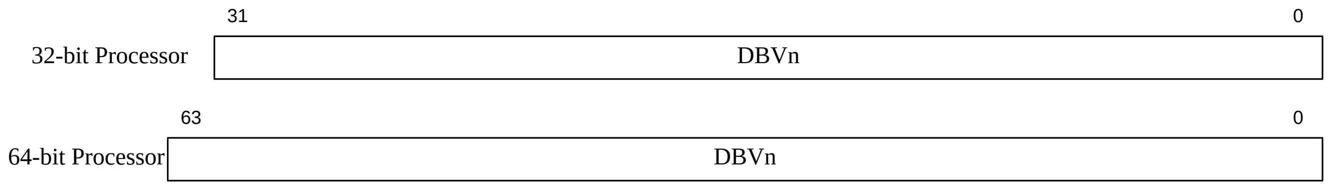


Table 3.23 DBVn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
DBV	MSB:0	Data breakpoint data value for condition. Debug software must adjust for endianess when programming this field.	R/W	Undefined	Required

3.8 Recommendations for Implementing Hardware Breakpoints

This section provides useful information for implementing instruction and data breakpoints.

3.8.1 Number of Instruction Breakpoints Without Single Stepping

If hardware single stepping is not implemented, then at least two instruction breakpoints are required. Four instruction hardware breakpoints are recommended.

3.8.2 Data Breakpoints with Data Value Compares

Data breakpoints should be implemented with data value compares. Also, data value compares should be implemented even if it is not possible to break on loads with precise data value compares. Refer to Section 3.8.4 on page 59 for more information on precise exceptions.

3.8.3 Data Breakpoint Compare on Invalid Data

Data breakpoints should only compare on valid data, meaning they only generate debug exceptions based on valid data in the compare. This does also apply to compare on store data for a store to an unaligned address. For example, no debug exception should be generated for a bus error on a load that has a pending data breakpoint to compare on the data returned by the load.

However, in some cases, the indication of invalid data is late relative to the data, for example, for a cache error as a result of a complex error detection. In this case, data breakpoints can indicate a debug exception because the data was believed to be valid at the time of the compare, and the pending error is then indicated to the debug handler through the DBusEP or CacheEP bit in the Debug register, because the error occurred after the debug exception.

However, for bus errors due to external events, the bus error indication usually is available when the compare in the data breakpoint would take place. Thus it is possible to avoid a debug exception.

3.8.4 Precise / Imprecise Debug Exceptions on Data Breakpoints with Data Value Compares

Data breakpoints are recommended to generate precise debug exceptions, if possible in the implementation. Thus the DEPC register and DBD bit in the Debug register point to the load/store that caused the debug exception to occur. This instruction can then be re-executed when execution resumes after the debug handler. However, data breakpoints are allowed to cause imprecise debug exceptions when the breakpoint is set up with data value compares; for example, if data breakpoints with compares on loaded data values cannot be made precise due to a non-blocking load. In this case, the DEPC register and DBD bit in the Debug register point to an instruction in the execution flow after the load/store that caused the imprecise debug exception. The BS bit can be updated when the match is detected, even though a debug exception is not taken until later due to internal stalls (for example, a nulled instruction in the pipeline at the time the match is detected). It is implementation specific as to which cases a data breakpoint can cause an imprecise debug exception. It is recommended that the data breakpoints cause imprecise matches in as few cases as possible.

In a processor implementing the MIPS MT ASE, since instructions from multiple thread contexts may be interleaved in the pipeline, imprecise data breakpoints is a bother since the thread taking the breakpoint exception may not be the thread that caused the breakpoint. Hence, it is required that in a processor implementing MIPS MT, the hardware must jam the value of the TC that caused the breakpoint in the TCval bits of the corresponding DBASIDn register. This must be done irrespective of whether or not the data breakpoint exception is implemented as a precise or an imprecise debug exception, for a consistent software implementation.

Implementations can require imprecise debug exceptions from data breakpoints on loads with value compares in a specific address range, if re-execution of a load in this range is not acceptable. This case is possible if the load has side effects such as removing an entry on a queue. Imprecise debug exceptions for value compares ensure that the destination register is properly updated with the loaded value, whereby re-execution of the load is avoided.

3.9 Breakpoint Examples

This section provides several examples of instruction and data breakpoint uses.

3.9.1 Instruction Breakpoint Examples

This section provides examples that illustrate using an instruction break.

3.9.1.1 Instruction Break in Small Range of Instructions with ASID

This example shows how to set up an instruction breakpoint to break on the fetch of any one of the four instructions in the virtual address range shown below:

```
0x0000 0010      J L1   // ASID = 0x5
0x0000 0014      NOP
0x0000 0018      J L2
0x0000 001C      NOP
```

The break registers must be set up as follows:

- IBA0 = 0x0000 0010
- IBM0 = 0x0000 000C
- IBC0: BE=1, ASIDuse=1, ASID = 0x5, other bits zero

Note that IBA0 has the starting address, and IBM0 has the address mask.

3.9.1.2 Instruction Break on 32-bit MIPS16e™ Instruction

In this example, instruction breakpoint 0 needs to be set up to break on the range 0x0000 0030 to 0x0000 0036, which starts with the second part of an extended MIPS16e instruction:

```
0x0000 002e      EXT    // (1st part of MIPS16e inst.)
0x0000 0030      ADD    // (2nd part)
0x0000 0032      SUB
0x0000 0034      SUB
0x0000 0036      SUB
```

The break registers must be set up as follows:

- IBA0 = 0x0000 0031
- IBM0 = 0x0000 0006
- IBC0: BE = 1, ASIDuse = 0, other bits zero

The CPU does not take a debug exception when fetching the second part of the ADD instruction, because it does not constitute a whole instruction. The first break is on the SUB instruction at 0x0000 0032.

3.9.2 Data Breakpoint

This section provides three examples of data breakpoints.

3.9.2.1 Data Break on Load Access with ASID

This example shows how to perform a break on data breakpoint 0 when the CPU loads data 0xAAAA 0000 from memory location 0x0000 0100 in ASID=0x7:

```
LW $2, 0x100($0)        // ASID = 0x7
```

The break registers must be set up as follows:

- DBA0 = 0x0000 0100
- DBM0 = 0x0
- DBV0 = 0xAAAA 0000
- DBC0: BE = 1, NoLB = 0, NoSB = 1, BLM = 0, BAI = 0, ASIDuse = 1, ASID = 0x7, other bits zero

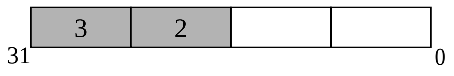
In this example, DBA0 contains the breakpoint address; DBM0 has the address mask; DBV0 has the data value; and DBC0 indicates a breakpoint condition might be fulfilled on a load but not on a store, there is a value compare for a corresponding byte, and an ASID is used.

3.9.2.2 Data Break on Store(s) to Halfword in Memory

This example shows a break on data breakpoint 0 when the CPU stores data in a specific halfword in memory. Stores to the other halfword at the same address can be ignored. The data word is illustrated in Figure 3.14; the halfword for bits 31:16 is shaded. The store instructions shown in Figure 3.14 alter the shaded halfword and cause a break if the breakpoint registers are set up as shown below.

Figure 3.14 Data Break on Store with Value Compare

Break on Memory Address 0x0000 0200 bit 31:16, Little Endian



```
SW        $2, 0x0000 0200  bytes_valid = 1111₂
SH        $2, 0x0000 0202  bytes_valid = 1100₂
SB        $2, 0x0000 0202  bytes_valid = 0100₂
SB        $2, 0x0000 0203  bytes_valid = 1000₂
```

In this example, the data breakpoint registers are set up as follows:

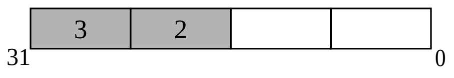
- DBA0 = 0x0000 0200
- DBM0 = 0
- DBC0: BE = 1, NoLB = 1, NoSB = 0, BLM = 1111_2, BAI = 0011_2, ASIDuse = 0, other bits zero

3.9.2.3 Data Break on Store(s) to Halfword Range in Memory with Certain Value

In this example, the most significant halfword in a given memory range is altered, and the most significant part of the halfword is written a certain value. The data word is illustrated below; the halfword for bits 31:16 is shaded. The store instructions shown in Figure 3.15 alter the shaded halfword and cause a break if the breakpoint registers are set up as shown below.

Figure 3.15 Data Break on Store with Value Compare

Break on Memory Address range 0x0000 0200 - 0x0000 02FC
Write to bits 31:16, bits 31:24 with value 0xAA, Little Endian



```
SW  $2, 0x0000 0220  $2=0xAAXX XXXX      bytes_valid = 1111₂
SH  $2, 0x0000 0242  $2=0xXXXX AAXX      bytes_valid = 1100₂
SB  $2, 0x0000 0282  $2=0xXXXX XXXX      bytes_valid = 0100₂
SB  $2, 0x0000 02F3  $2=0xXXXX XXAA      bytes_valid = 1000₂
'X' denotes undefined value.
```

In this example, the data breakpoint registers are set up as follows:

- DBA0 = 0x0000 0200
- DBM0 = 0x0000 00FC

- DBV0 = 0xAA00 0000
- DBC0: BE = 1, NoLB = 1, NoSB = 0, BLM = 0111_2, BAI = 0011_2, ASIDuse = 0, other bits zero

Chapter 4

Complex Break and Trigger Block

The complex break and trigger (CBT) block is part of the EJTAG breakpoint block and consequently integrated into the core logic when implemented. The CBT block is optional and defined in EJTAG spec 4.00 and above. A bit in the EJTAG DCR$_{CBT}$ register (bit 10) indicates the presence of this CBT block in the implementation. The CBT block implements complex breakpoint matching conditions that include: match primed by previous breakpoint match, qualified by previous data break match, matched using pass-counters, matches enabled by the **and** of two other break matches, etc.

The CBT block provides enhanced breakpoint and trace control capability based on the standard instruction and data breakpoints.

4.1 Complex Trigger Features/Capabilities

The complex trigger unit will be integrated into the existing EJTAG break unit. All of the previous simple break features will be preserved. This section describes the enhancements to be included in the complex trigger block.

Note: the term breakpoints in this section refers to either actual breakpoints that take a debug exception or trigger points that only record the status and send this signal to the trace block.

- Pass Counters - each break channel has a counter associated with it that enables a breakpoint to only be taken after the address/value condition has been met a certain number of times.
- Data Qualified breakpoints - these can be enabled and disabled based on the state of a data breakpoint condition which can be used to only match on instructions executed in a certain process.
- Primed breakpoints - these are only enabled once another breakpoint has occurred which allows breaking on a simple sequences of events.
- Stopwatch timer - a counter that can be configured to start or stop based on specific instruction breakpoints.
- Ability to support ‘tuples’ - breakpoints that only fire when both instruction and data conditions match on a single instruction.

4.2 General Complex Break Behavior

There is some general complex break behavior that is common to all the features. This behavior is described below:

- Resets to a disabled state when the core is reset. The complex break functionality will be disabled and debug software that is not aware of complex break should continue to function normally.
- Complex break state is not updated on exceptional instructions.

- Complex breakpoints should be implemented such that there is no hazard between enabling and enabled events. When an instruction causes an enabling event, the following instruction sees the enabled state and reacts accordingly.

4.3 Registers in the Complex Break and Trigger Block

The CBTC (complex break and trigger control) register indicates the specific implementation choices made from the architecture specification. The complex break and trigger block also adds new control registers for the complex control for Instruction and Data breaks. These registers are IBCCn and DBCCn, where n is the number of implemented instruction or data breaks, to a maximum possible value of 15. The drseg addresses for all these registers are shown in Table 4.1.

Table 4.1 Registers in the Complex Break and Trigger Block and Their drseg Memory Addresses

Register Mnemonic	drseg Address Offset	Description
CBTC	0x8000	Complex Break and Trigger Control (see Figure 4.1)
IBCCn	0x1120 + 0x100 * n	Instruction Breakpoint Complex Control n (see Figure 4.2)
IBPCn	0x1128 + 0x100 * n	Instruction Breakpoint Pass Counter n (see Figure 4.3)
DBCCn	0x2128 + 0x100 * n	Data Breakpoint Complex Control n (see Figure 4.4)
DBPCn	0x2130 + 0x100 * n	Data Breakpoint Pass Counter n (see Figure 4.5)
PrCndAIn	0x8300 + 0x20*n	Prime Condition Register A for Instruction breakpoint n (see Figure 4.6)
PrCndADn	0x84E0 + 0x20*n	Prime Condition Register A for Data breakpoint n (see Figure 4.6)
STCtl	0x8900	Stopwatch Timer Control (see Figure 4.7)
STCnt	0x8908	Stopwatch Timer Count (see Figure 4.8)

4.3.1 Complex Break and Trigger Control (CBTC) Register (0x8000)

Compliance Level: Implemented only if complex breakpoints are implemented.

The CBTC register contains configuration bits that indicate which features of complex break are implemented as well as a control bit for the stopwatch timer. It is possible for an implementation to implement complex breaks and implement any non-zero subset of these features. Figure 4.1 shows the format of the CBTC register; Table 4.2 describes the CBTC register fields.

Figure 4.1 CBTC Register Format

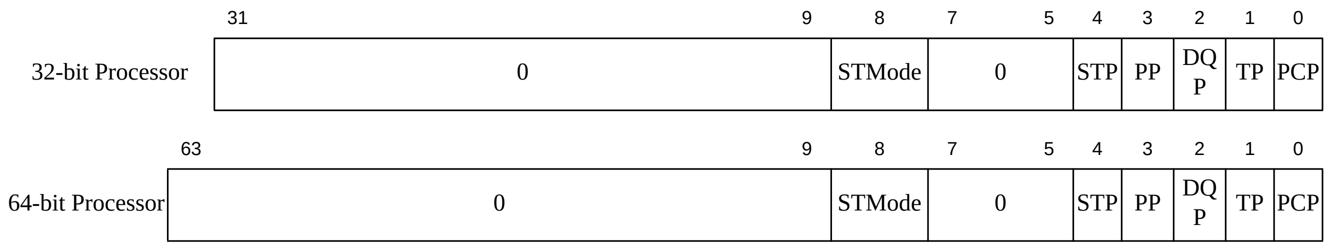


Table 4.2 CBTC Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
STMode	8	Indicates the current operating mode of the stopwatch timer, provided this is present as indicated by bit STP: **Encoding** / **Meaning** 0 / Is in free-running mode 1 / Is activated by specified break pairs	R/W	1	Required if complex break is present (DCR_{CBT} = 1)
STP	4	Indicates if the stopwatch timer is implemented. This is optional if complex breaks feature is present: **Encoding** / **Meaning** 0 / No stopwatch timer present 1 / Stopwatch timer is present	R	Preset	Required if complex break is present (DCR_{CBT} = 1)
PP	3	Indicates if primed breakpoints are implemented This is optional if complex breaks feature is present: **Encoding** / **Meaning** 0 / No primed breaks are present 1 / Primed breaks are present	R	Preset	Required if complex break is present (DCR_{CBT} = 1)
DQP	2	Indicates if data qualified breakpoints are implemented This is optional if complex breaks feature is present: **Encoding** / **Meaning** 0 / No data qualified breaks present 1 / Data qualified breaks are present	R	Preset	Required if complex break is present (DCR_{CBT} = 1)
TP	1	Indicates if tuple breakpoints are implemented This is optional if complex breaks feature is present: **Encoding** / **Meaning** 0 / No tuples breaks present 1 / Tuple breaks are present	R	Preset	Required if complex break is present (DCR_{CBT} = 1)
PCP	0	Indicates if the pass counter feature is implemented This is optional if complex breaks feature is present: **Encoding** / **Meaning** 0 / Do not use it as triggerpoint 1 / Use it as triggerpoint	R	Preset	Required if complex break is present (DCR_{CBT} = 1)
0	MSB:9, 7:5	Must be written as zeros; return zeros on read.	R	0	Reserved

Each instruction and data breakpoint now have two additional registers as shown in Table 4.1.

4.3.2 Instruction Breakpoint Complex Control n (IBCCn) Register (0x1120 + n * 0x100)

Compliance Level: Implemented only if complex breakpoints are implemented and only for implemented instruction breakpoints.

The Instruction Breakpoint Complex Control n (IBCCn) register controls the complex break conditions for instruction breakpoint n. Figure 4.2 shows the format of the IBCCn register; Table 4.3 describes the IBCCn register field.

Figure 4.2 IBCCn Register Format

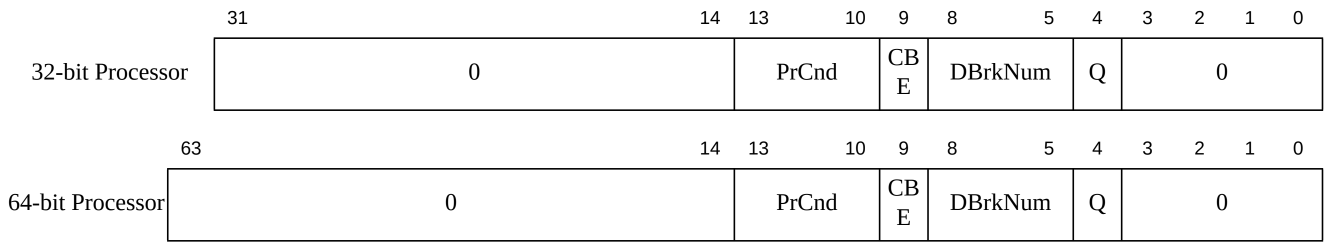


Table 4.3 IBCCn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	**Bits**				
PrCnd	13:10	Specifies the priming condition for I breakpoint n. The architecture allows for up to 16 priming conditions to choose from, where the 0000 value specifies the default bypass mode of no priming condition. An implementation can choose to define from no priming condition (default bypass mode) to up to 15 other possible priming conditions. These 15 priming condition values are specified in up to 4 priming condition registers per breakpoint (A/B/C/D). See Section 4.3.6 on page 69.	R/W	0	Required if primed breaks are present ($CBTC_{PP}$ = 1)
CBE	9	Complex break enable bit is used to indicate that this breakpoint may be used in a complex sequence which includes: as a priming condition for another breakpoint, to start or stop the stopwatch timer, or as part of a tuple breakpoint.	R/W	0	Required
DBrkNum	8:5	Indicates which data breakpoint channel is used to qualify this instruction breakpoint	R	Preset	Required if data qualified breaks are present ($CBTC_{DQP}$ = 1)
Q	4	Qualify this breakpoint based on the data breakpoint indicated in DBrkNum: **Encoding** / **Meaning** 0 — Not dependent on qualification 1 — Breakpoint must be qualified to be taken	R/W	0	Required if data qualified breaks are present ($CBTC_{DQP}$ = 1)
0	MSB:14, 3:0	Must be written as zeros; return zeros on read.	R	0	Reserved

4.3.3 Instruction Breakpoint Pass Counter n (IBPCn) Register (0x1128 + n*0x100)

Compliance Level: Implemented only if complex breakpoints are implemented and only for implemented instruction breakpoints.

The Instruction Breakpoint Pass Counter n (IBPCn) register controls the pass counter associated with instruction breakpoint n. The width of the actual counter is implementation-dependent. To determine the width software can write a value of -1 to the register and read back the value to note the bits that were set on the write. Figure 4.3 shows the format of the IBPCn register; Table 4.4 describes the IBPCn register field.

Figure 4.3 IBPCn Register Format

	31 … n+1	n … 0
32-bit Processor	0	PassCnt

	63 … n+1	n … 0
64-bit Processor	0	PassCnt

Table 4.4 IBPCn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
PassCnt	n:0	For the breakpoint associated with this pass counter, each time the matching condition is seen, this value will be decremented by 1. When the value reaches 0, or was originally set to 0, subsequent matches will cause a break or trigger as requested and the counter will stay at 0. Note that when the pass counter value is greater than 0, a break/trigger action will never be taken even on a matching condition. The only action taken would be to decrement the pass counter by 1. The instruction pass counter should not be set on instruction breakpoints that are being used as part of a tuple breakpoint.	R/W	0	Required if pass counters are present ($CBTC_{PCP}$ = 1)
0	MSB:n+1	Must be written as zeros; return zeros on read.	R	0	Reserved

4.3.4 Data Breakpoint Complex Control n (DBCCn) Register (0x2128 + n * 0x100)

Compliance Level: Implemented only if complex breakpoints are implemented and only for implemented data breakpoints.

The Data Breakpoint Complex Control n (DBCCn) register controls the complex break conditions for data breakpoint n. Figure 4.4 shows the format of the DBCCn register; Table 4.5 describes the DBCCn register field.

Figure 4.4 DBCCn Register Format

	31 … 18	17 … 14	13 … 10	9	8 … 5	4	3 2 1 0
32-bit Processor	0	UnPrCnd	PrCnd	CBE	DBrkNum	Q	0

	31 … 18	17 … 14	13 … 10	9	8 … 5	4	3 … 0
	63 … 18	17 … 14	13 … 10	9	8 … 5	4	3 … 0
64-bit Processor	0	UnPrCnd	PrCnd	CBE	DBrkNum	Q	0

Table 4.5 DBCCn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
UnPrCnd	17:14	Specifies the unpriming condition for I breakpoint n. This field simply points to one of the 16 architecturally defined priming conditions. This condition is then considered to unprime this I breakpoint. The 0000 value specifies the default bypass mode of no unpriming condition, and an implementation may choose to tie this field to a zero value and make this field not writeable and hence disallow software to specify an unprime condition. The remaining 15 unpriming condition values are specified in up to 4 priming condition registers per breakpoint (A/B/C/D). See Section 4.3.6 on page 69.	R/W	0	Required if primed breaks are present ($CBTC_{PP}$ = 1)
PrCnd	13:10	Specifies the priming condition for I breakpoint n. The architecture allows for up to 16 priming conditions to choose from, where the 0000 value specifies the default bypass mode of no priming condition. An implementation can choose to define from no priming condition (default bypass mode) to up to 15 other possible priming condition. These 15 priming condition values are specified in up to 4 priming condition registers per breakpoint (A/B/C/D). See Section 4.3.6 on page 69.	R/W	0	Required if primed breaks are present ($CBTC_{PP}$ = 1)
CBE	9	Complex break enable bit is used to indicate that this breakpoint may be used in a complex sequence which includes: as a priming condition for another breakpoint, to start or stop the stopwatch timer, or as part of a tuple breakpoint.	R/W	0	Required
DBrkNum	8:5	Indicates which data breakpoint channel is used to qualify this instruction breakpoint	R	Preset	Required if data qualified breaks are present ($CBTC_{DQP}$ = 1)
Q	4	Qualify this breakpoint based on the data breakpoint indicated in DBrkNum: **Encoding** / **Meaning** 0 / Not dependent on qualification 1 / Breakpoint must be qualified to be taken	R/W	0	Required if data qualified breaks are present ($CBTC_{DQP}$ = 1)
0	MSB:14, 3:0	Must be written as zeros; return zeros on read.	R	0	Reserved

4.3.5 Data Breakpoint Pass Counter n (DBPCn) Register (0x2130 + n*0x100)

Compliance Level: Implemented only if complex breakpoints are implemented and only for implemented data breakpoints.

The Data Breakpoint Pass Counter n (DBPCn) register controls the pass counter associated with data breakpoint n. The width of the actual counter is implementation-dependent. To determine the width software can write a value of -1 to the register and read back the value to note the bits that were set on the write. Figure 4.5 shows the format of the DBPCn register; Table 4.6 describes the DBPCn register field.

Figure 4.5 DBPCn Register Format

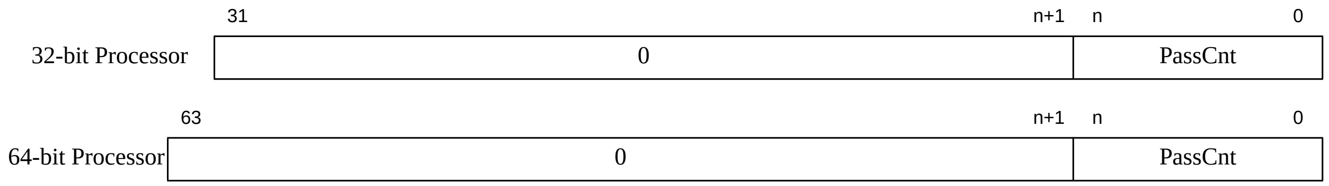


Table 4.6 DBPCn Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
PassCnt	n:0	For the breakpoint associated with this pass counter, each time the matching condition is seen, this value will be decremented by 1. When the value reaches 0, or was originally set to 0, subsequent matches will cause a break or trigger as requested and the counter will stay at 0. Note that when the pass counter value is greater than 0, a break/trigger action will never be taken even on a matching condition. The only action taken would be to decrement the pass counter by 1. The data pass counters are re-used for a tuple breakpoint that may be currently associated with the data break.	R/W	0	Required if pass counters are present ($CBTC_{PCP}$ = 1)
0	MSB:n+1	Must be written as zeros; return zeros on read.	R	0	Reserved

4.3.6 Priming Condition A I/D n (PrCndAI/Dn) Registers

Compliance Level: Implemented if complex breakpoints are implemented.

The Prime Condition registers hold implementation specific information on which trigger points are used for the priming conditions for each breakpoint register. These priming conditions are predetermined by implemented and cannot be changed dynamically by software, hence these registers are read-only.

The architecture allows up to 16 priming conditions per breakpoint, and there can be up to 4 priming condition registers per breakpoint (A/B/C/D) that contains the necessary information for all 16 priming conditions. An implementation only needs to implement as many priming conditions and registers as there are needed to support the number of implemented priming conditions. Each register hold the information for 4 priming conditions.

Figure 4.5 shows the format of the PrCndA register; Table 4.6 describes the PrCndA register field. This register is identical for both Instruction and Data and define the first 4 priming conditions. The other three registers PrCndB, PrCndC, and PrCndD are similar and implement the remaining 12 possible conditions. Each condition CondN in the register specified which trigger point is connected to priming condition 0 through 15 for the current breakpoint. Note that condition 0 is always Bypass and will read the 8 priming condition bits as 8’b0.

Figure 4.6 PrCndA Register Format

	31 … 24	23 … 16	15 … 8	7 … 0
32-bit Processor	Cond3	Cond2	Cond1	Cond0

Table 4.7 PrCndA Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
CondN	31:30 23:22 15:14 7:6	Reserved	R	0	Required if priming conditions are present ($CBTC_{PP}$ = 1)
	29:28 21:20 13:12 5:4	Trigger type 00 - Special/Bypass 01 - Instruction 10 - Data 11 - Reserved	R	Preset	
	27:24 19:16 11:8 3:0	Break Number, 0-14	R	Preset	

4.3.7 Stopwatch Timer Control (STCtl) Register (0x8900)

Compliance Level: Implemented if stopwatch timer is implemented.

The Stopwatch Timer Control register gives configuration information about how the stopwatch timer register is controlled. Figure 4.7 shows the format of the STCtl register; Table 4.8 describes the STCtl register field.

Figure 4.7 STCtl Register Format

	31 … 22	21	20	19	18	17 … 14	13 … 10	9	8 … 5	4 … 1	0
32-bit Processor	0	BTStop1	BTStart1	BTStop0	BTStart0	StopChan1	StartChan1	En1	StopChan0	StartChan0	En0

	63 … 22	21	20	19	18	17 … 14	13 … 10	9	8 … 5	4 … 1	0
64-bit Processor	0	BTStop1	BTStart1	BTStop0	BTStart0	StopChan1	StartChan1	En1	StopChan0	StartChan0	En0

Table 4.8 STCtl Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
BTStop1	21	Break type for Stop Channel 1. A value of 0 implies instruction and 1 implies data (this could be a tuple if the data is currently part of a tuple). An implementation that ties the start and stop channels to predefined breakpoints will also tie this value to a predefined value.	R/W	x	
BTStart1	20	Break type for Start Channel 1. A value of 0 implies instruction and 1 implies data (this could be a tuple if the data is currently part of a tuple). An implementation that ties the start and stop channels to predefined breakpoints will also tie this value to a predefined value.	R/W	x	
BTStop0	19	Break type for Stop Channel 0. A value of 0 implies instruction and 1 implies data (this could be a tuple if the data is currently part of a tuple). An implementation that ties the start and stop channels to predefined breakpoints will also tie this value to a predefined value.	R/W	x	
BTStart0	18	Break type for Start Channel 0. A value of 0 implies instruction and 1 implies data (this could be a tuple if the data is currently part of a tuple). An implementation that ties the start and stop channels to predefined breakpoints will also tie this value to a predefined value.	R/W	x	
StopChan1	17:14	Indicates the breakpoint channel for the second pair that will stop the counter if the timer is under breakpoint control. An implementation can choose to tie this to a predefined breakpoint. But it is possible for implementation to allow this field to be writable by software so that the pair of start and start channels is dynamically selectable.	R/W	x	Optional
StartChan1	13:10	Indicates the breakpoint channel for the second pair that will start the counter if the timer is under breakpoint control. An implementation can choose to tie this to a predefined breakpoint. But it is possible for implementation to allow this field to be writable by software so that the pair of start and start channels is dynamically selectable.	R/W	x	
En1	9	Enable the second pair of breakpoint registers to control the timer under breakpoint control.	R/W	x	
StopChan0	8:5	Indicates the breakpoint channel that will stop the counter if the timer is under breakpoint control. An implementation can choose to tie this to a predefined breakpoint. But it is possible for implementation to allow this field to be writable by software so that the pair of start and start channels is dynamically selectable.	R/W	x	Required if stopwatch timer is present (CBTC$_{STP}$ = 1)
StartChan0	4:1	Indicates the breakpoint channel that will start the counter if the timer is under breakpoint control. An implementation can choose to tie this to a predefined breakpoint. But it is possible for implementation to allow this field to be writable by software so that the pair of start and start channels is dynamically selectable.	R/W	x	
En0	0	Enable the first pair of breakpoint registers to control the timer under breakpoint control.	R/W	x	

Table 4.8 STCtl Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
0	MSB:22	Must be written as zero; returns zero on read.	R	0	Reserved

4.3.8 Stopwatch Timer Count (STCnt) Register (0x8908)

Compliance Level: Implemented if stopwatch timer is implemented.

The Stopwatch Timer Count register is the count value for the stopwatch timer. Figure 4.8 shows the format of the STCnt register; Table 4.9 describes the STCnt register field.

Figure 4.8 STCnt Register Format

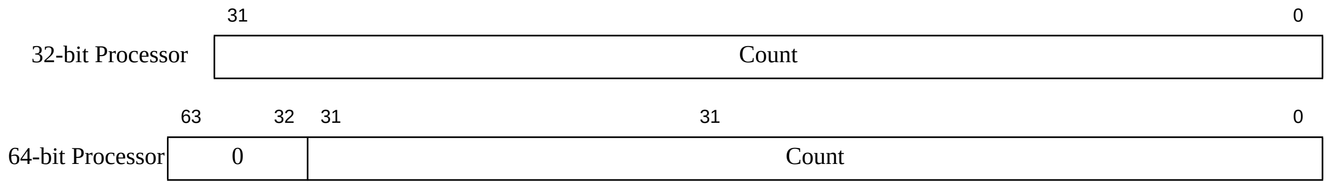


Table 4.9 STCnt Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
Count	31:0	Current counter value	R/W	0	Required if stopwatch timer is present ($CBTC_{STP}$ = 1)

4.4 Tuple Breakpoints

A tuple breakpoint is the logical AND of a data breakpoint and an instruction breakpoint. Whether or not this feature is present is indicated by $CBTC_{TP}$. Tuple breakpoints are specified as a condition on a data breakpoint. In the data breakpoint complex control register, if the TUP bit is set ($DBCCn_{TUP}$), the data breakpoint will not match unless the corresponding instruction breakpoint is set up for a tuple and the matching conditions are also met. The instruction breakpoint must be set up as follows to be considered part of a tuple breakpoint:

- $IBCCn_{CBE}$ -= 1
- $IBCCn_{PrCnd} = IBCCn_{DQ} = IBCn_{TE} = IBCn_{BE} = IBPCn = 0$

Note that if the instruction breakpoint has BreakEnable set, the instruction will take a simple instruction breakpoint, and if it is precise, the instruction will not be executed and the data side of the tuple will not even be evaluated.

A tuple uses the data breakpoint resources to specify the break action, break status, pass counter, data qualifier, and priming conditions.

4.5 Pass Counters

Pass counters are used to specify that the breakpoint conditions must match N times before the breakpoint action will be enabled, where N is the value written by software to a pass counter register. Whether or not this feature is present is indicated by $CBTC_{PCP}$. The pass counter registers are drseg memory-mapped and added for each instruction and data break channel, see Section 4.3.3 "Instruction Breakpoint Pass Counter n (IBPCn) Register (0x1128 + n*0x100)" and Section 4.3.5 "Data Breakpoint Pass Counter n (DBPCn) Register (0x2130 + n*0x100)". The data breakpoint pass counter registers are reused for tuple breakpoints. Pass counter usage is specified below:

- The architecture allows implementation to implement up to as many pass counters as the number of implemented instruction and data breakpoints.
- The width of the counter is implementation dependent. Software can determine the width and presence of a counter by writing a value of -1 to the register and reading back to see which bits are set. When no bits are set, this implies that this breakpoint does not implement a pass counter. The recommended counter size is 8 bits for instruction breakpoints and 16 bits for data breakpoints.
- Writing a non-zero value to this register will enable the pass counters. When enabled, each time the breakpoint conditions match, the counter will be decremented by 1. Once the counter value reaches 0, the breakpoint action (breakpoint exception, trigger, or complex break enable) will occur on any subsequent matches and the counter will not decrement further.
- If the breakpoint also has priming conditions and/or data qualifier specified, the pass counter will only decrement when the priming and/or data qualifier conditions have been met. A breakpoint condition can be changed from being qualified to unqualified without any affect on the counter state.
- If a data breakpoint is configured to be a tuple breakpoint, the data pass counter will only decrement on instructions where both the instruction and data break conditions match. The pass counter for the instruction break involved in a tuple should not be enabled if the tuple is enabled.
- Writing a value of 0 to the counter will disable the pass counter and enable the breakpoint to fire whenever the conditions are met. The counter is reset to 0 to preserve compatibility with legacy software.
- The counter register will be updated as matches are detected, and the current value can be read from the register while operating in debug mode. It is not a requirement, but an architectural recommendation that the current count value be reflected in the drseg register that represents the counter.
- In some implementations, a simple instruction breakpoint may be taken precisely, while a complex favor like the one that uses pass counters may be taken imprecisely. In this situation, when a complex condition like pass counters is disabled during execution, the breakpoint exceptions will continue to be taken imprecisely until the complex condition is cleared, for example, the pass counter is actually written with the zero value.

4.6 Data Qualified Breakpoints

Each of the breakpoints, instruction, data, or tuple can be data qualified. Whether or not this feature is present is indicated by $CBTC_{DQ}$. In qualified mode, a breakpoint will recognize its conditions only after the specified data breakpoint matches both address and data. If the qualifying data breakpoint matches the address but has a mismatch on the data value, the breakpoint with the qualifier will be disqualified and will not match until a subsequent qualifying match.

Which data break can qualify another break, be it instruction, data, or tuple breakpoint is predetermined by the implementation hardware and preset in register bits IBCCn$_{DBrkNum}$ and DBCCn$_{DBrkNum}$. The IBCCn$_{Q}$ and DBCC$_{Q}$ bits are used by software to decide when an instruction or data breakpoint respectively should be actively considered to be data qualified. See Section 4.3.2 "Instruction Breakpoint Complex Control n (IBCCn) Register (0x1120 + n * 0x100)" and Section 4.3.4 "Data Breakpoint Complex Control n (DBCCn) Register (0x2128 + n * 0x100)". The tuple breakpoint reuses the bits in the corresponding DBCCn register of the data breakpoint that forms the tuple.

This feature can be used similarly to the ASID qualification that is available on cores with TLBs. If an RTOS loads a process ID for the current process, that load can be used as the qualifying breakpoint. When a matching process ID is loaded (entering the desired RTOS process), qualified instruction breakpoints will be enabled. When a different process IS is loaded (leaving the desired RTOS process), the qualified instruction breakpoints are disabled. Alternatively, with the InvertValueMatch feature of the data breakpoint, the instruction breakpoints could be enabled on a any process ID other than the specified one.

Enabling the data qualifier requires the following to be true:

- Qualifier (data break) must have DBCn$_{TE}$ or DBCCn$_{CBE}$ set.
- Qualifier should have data comparison enabled (via settings of DBCn$_{BLM}$ and DBCn$_{BAI}$).
- Qualifier should not have pass counters, priming conditions, or tuples enabled.
- Qualifier can be either a load or store instruction.

4.7 Primed Breakpoints

Priming conditions provide a way for one breakpoint to be enabled by another one. Whether or not this feature is present is indicated by CBTC$_{PP}$. Prior to the priming condition being satisfied, any breakpoint matches are ignored. It is possible for a primed breakpoint to get unprimed. Once unprimed, the breakpoint must be primed again before a matching condition will enable the breakpoint to take a break or trigger action. The details of this feature are:

- Each breakpoint has a choice of up to a maximum of 16 possible priming conditions. An implementation may limit this to a smaller number and will list the specific priming conditions for each of its breakpoints for reference. The priming conditions vary from breakpoint to breakpoint (since it makes no sense for a breakpoint to prime itself).
- Each Prime condition is the comparator output after it has been qualified by its own Prime condition and pass counter. Using this, several stages of Priming are possible (e.g. data cycle D followed by instruction A followed by instruction B followed by instruction C).
- One of the conditions is a bypass mode in which the priming condition is always met. This bypass condition is the default state of a breakpoint and initialized on reset to be backwards compatible to the simple instruction and data breakpoints.
- The priming breakpoint must have IBCn$_{TE}$ or IBCCn$_{CBE}$ set if it is an instruction breakpoint, or it must have DBCn$_{TE}$ or DBCCn$_{CBE}$ set if it is a data (or tuple) breakpoint.
- The IBCCn$_{UnPrCnd}$ and DBCCn$_{UnPrCnd}$ are used to specify a condition used to unprime the instruction or data breakpoint respectively. This is optional since an implementation can tie this field to 0 and disallowing software to write to this field. This implies that the unprime feature is a bypass and it is not possible to unprime a breakpoint once it is primed. A breakpoint is considered to start in the unprimed condition until it matches a priming

condition. Encountering an unprime condition match will take the breakpoint to the unprime state if it was primed, or leave it unprimed if it was already in the unprimed state.

Section 4.3.6 "Priming Condition A I/D n (PrCndAI/Dn) Registers" shows the registers used to indicate the prime or unprime condition. The full table of all the PrCnd Registers and their drseg addresses is shown in Table 4.10.

Table 4.10 Addresses for PrCnd[A-D][I/D]N Registers in drseg Memory

Register	drseg Address	Reset value
PrCndAI0	0x8300	Preset
PrCndBI0	0x8308	Preset
PrCndCI0	0x8310	Preset
PrCndDI0	0x8318	Preset
PrCndAI1	0x8320	Preset
...	Block of 3 addresses	Preset
PrCndAI2	0x8340	Preset
...	Block of 3 addresses	Preset
PrCndAI3	0x8360	Preset
...	Block of 3 addresses	Preset
PrCndAI4	0x8380	Preset
...	Block of 3 addresses	Preset
PrCndAI5	0x83A0	Preset
...	Block of addresses	Preset
PrCndIA14	0x84C0	Preset
...	Block of 3 addresses	Preset
PrCndAD0	0x84E0	Preset
PrCndBD0	0x84E8	Preset
PrCndCD0	0x84F0	Preset
PrCndDD0	0x84F8	Preset
PrCndAD1	0x8500	Preset
...	Block of addresses	Preset
PrCndAD14	0x86A0	Preset
...	Block of 3 addresses	Preset

The architecture does not restrict implementation as to when the primed, qualified, or tuple breakpoints are recognized and hence also when the pass counter update occurs. In the current EJTAG specification, simple instruction breaks are expected to be precise, that is, recognized early in the pipe and later fetches are squashed as soon as possible. (Nevertheless, note that the actual break exception is taken only after the instruction passes the point of other possible exceptions in the pipe). Data breaks, on the other hand, may be precise or imprecise. If imprecise, then they are not recognized until later in the pipe and hence early squashing of fetches is not possible. In the presence of complex breaks which may be recognized late in the pipe (later than simple instruction breaks), an instruction break of a later instruction may be primed by a data break from an earlier instruction in the execution sequence, because of the different pipeline stages when these breaks may be recognized. This causes a hazard condition. Although it may not be possible to entirely remove this hazard with complex breaks, its effect on implementation complexity may be reduced by allowing all complex breaks to be recognized later in the pipe and the pass counter update is also done later in the pipe. This reduces the need for speculative updates of the pass counter and roll backs of state when the instruction may be squashed for other reasons. Given this type of complex interaction in the pipeline, the architecture recommends that the recognition of simple instruction breaks be retained at the early pipe stages, while all complex break recognition be delayed to the stage where the data breaks are recognized.

4.8 Stopwatch Timer

The stopwatch timer is a count register that is drseg memory-mapped so that it can be read and reset by software, see Section 4.3.8 "Stopwatch Timer Count (STCnt) Register (0x8908)". The presence of this feature is indicated by bit CBTC_{STP}. A stopwatch control register is used to control its operation, see Section 4.3.7 "Stopwatch Timer Control (STCtl) Register (0x8900)". The stopwatch timer works as follows:

- Count value is reset to 0.
- The timer can be configured to be in a free running mode or controlled to start and stop by specific breakpoints using $\text{CBTC}_{\text{STMode}}$.
- The ability to start and stop the timer using breakpoints can be a useful feature, or example, using instruction breaks to start and stop the timer, it would be possible to measure the amount of time spent in a particular body of code by setting the start break channel to point to the entry point and the stop break channel to point to the exit point.
- The architecture allows up to two pairs of start/stop break channels. An implementation can choose to only implement one pair. If the stopwatch timer feature is implemented, then at least one pair of start/stop breakpoints must be implemented.
- Reset state has counter stopped and under breakpoint control so that the counter is not running when the core is not being actively debugged.
- The counter stops counting on entry into debug mode.
- When controlled by breakpoints, the controlling breakpoints should have the corresponding IBCn_{TE} or IBCn_{CBE} bit set for instructions breaks and the bit set for data (or tuple) breaks.
- The architecture allows software to program the start and stop hardware breakpoints, but an implementation can choose to predetermine these breakpoints, only allowing software the ability to enable one pair or the other. Software must write -1 to the STCtl register and read back the value to determine whether or not an implementation has provided software with the ability to program the start/stop breaks and how many pairs are implemented.
- Note that if two pairs are implemented, then enabling both will cause the hardware to use pair 0 as the controlling pair.

4.9 Reporting of the Complex Breakpoints in the Debug Register

Described here are the changes to the Debug register (number 23, select 0) and a new CP0 register Debug2 (number 23, select 6) which are used to define the cause of debug breaks when the cause arises from a complex breakpoint.

4.9.1 Debug Register (23, select 0) Changes for Complex Breakpoints

The Debug register now defines a field DIBImpr, which may be described as an indicator that a Debug Instruction Break exception occurred on an instruction due to an imprecise instruction hardware break.

4.9.2 Debug2 Register (23, select 6)

This is a new CP0 register specifically for use by the EJTAG block. The currently defined bits in this new register are defined in Section 6.7.2 "Debug2 Register (CP0 Register 23, Select 6)".

The expected bits to be set on a complex break implementation where all the complex breaks are taken imprecisely is as shown in the table below. Note that this does not imply anything about the simple breaks. The simple breaks could be taken precisely or imprecisely as per the implementation methodology.

Table 4.11 Debug Break Indicator Bits Set for Simple and Complex Breaks

Breakpoint Type	Debug Register Bits Set	Debug2 Register Bits Set
Simple Precise Ibreak	DIB	-
Simple Precise Dbreak	DDBL or DDBS	-
Simple Imprecise Ibreak	DIBImpr	-
Simple Imprecise Dbreak	DDBLImpr or DDBSImpr	-
Complex Tuple Break Imprecise	DIBImpr and (DDBLImpr or DDBSImpr)	Tup
Complex Data Qualified Ibreak Imprecise	DIBImpr	DQ
Complex Data Qualified Dbreak Imprecise	DDBLImpr or DDBSImpr	DQ
Complex Data Qualified Tuple Break Imprecise	DIBImpr and (DDBLImpr or DDBSImpr)	DQ and Tup
Complex Primed Ibreak Imprecise	DIBImpr	Prm
Complex Primed Dbreak Imprecise	DDBLImpr or DDBSImpr	Prm
Complex Primed Tuple break Imprecise	DIBImpr and (DDBLImpr or DDBSImpr)	Tup and Prm
Complex Ibreak with Pass Counter Imprecise	DIBImpr	PaCo
Complex Dbreak with Pass Counter Imprecise	DDBLImpr or DDBSImpr	PaCo
Complex Tuple Break with Pass Counter Imprecise	DIBImpr and (DDBLImpr or DDBSImpr)	Tup and PaCo
Complex Data Qualified Ibreak with Pass Counter Imprecise	DIBImpr	DQ and PaCo
Complex Data Qualified Dbreak with Pass Counter Imprecise	DDBLImpr or DDBSImpr	DQ and PaCo
Complex Data Qualified Tuple Break with Pass Counter Imprecise	DIBImpr and (DDBLImpr or DDBSImpr)	DQ and Tup and PaCo

Table 4.11 Debug Break Indicator Bits Set for Simple and Complex Breaks

Breakpoint Type	Debug Register Bits Set	Debug2 Register Bits Set
Complex Primed Ibreak with Pass Counter Imprecise	DIBImpr	Prm and PaCo
Complex Primed Dbreak with Pass Counter Imprecise	DDBLImpr or DDBSImpr	Prm and PaCo
Complex Primed Tuple Break with Pass Counter Imprecise	DIBImpr and (DDBLImpr or DDBSImpr)	Prm and Tup and PaCo

Chapter 5

PC Sampling

This chapter describes the optional PC sampling feature of EJTAG which is being introduced in the 3.1 version of the EJTAG specification. It contains the following sections:

5.1 Introduction

It is often useful for program profiling and analysis purposes to sample the value of the PC periodically. This information can be used for statistical profiling of the program akin to gprof. This information is also very useful for detecting hot-spots in the code. In a multi-threaded environment, this information can be used to understand thread behavior and verify thread scheduling mechanisms in the absence of a full-fledged tracing facility like PDtrace. The rest of this chapter describes the PC sampling feature. The PC sampling feature is optional within EJTAG. But EJTAG and the TAP controller must be implemented if PC Sampling is required. When implemented, PC sampling can be turned on or off, that is, it can be enabled or disabled. But when enabled, the PC value is continually sampled.

5.2 Overview of the PC Sampling Feature

The presence or absence of the PC Sampling feature is known from bit 9 (PCS) in the Debug Control register. As mentioned already, if PC sampling is implemented, and the PCSe (PC Sample Enable) bit 5 in the Debug Control Register is set to one, then the PC values are constantly sampled at the requested rate. The sampled PC values are written into a TAP register. The old value in the TAP register is overwritten by a new value even if this register has not been read out by the debug probe.

The sample rate is specified in a manner similar to the PDtrace synchronization period, with three bits. These bits in the Debug Control register (DCR) are 8:6 and called PCSR (PC Sample Rate). These three bits take the value 2^5 to 2^{12} similar to SyncPeriod. A write to these bits to modify the PCSR must reset the internal PC sample counter so that counting for the requested sample rate is immediately restarted and not wait until after the previous count has completed. For example, assume that the current sample period is 4096 cycles and the processor is counting up to this sample. If the user then wants to modify the sample rate to 256 cycles, then the processor must not wait for the entire 4096 cycles before beginning the new count, the counter is immediately reset to zero to begin the new count. Note that if a given implementation counts down and not up, then the internal counter must be reset immediately to not zero, but the newly requested sample rate to begin the count down.

The sampled values includes a New data bit, the PC, the ASID of the sampled PC as well as the Thread Context id if the processor implements MIPS MT. Figure 5.1 shows the format of the sampled values in the TAP register PCsample. The new data bit is used by the probe to determine if the PCsample register data just read out is new or already been read and must be discarded.

Figure 5.1 TAP Register PCsample Format

48 41	40 33	32 1	0
TC (for MIPS MT processors only)	ASID	PC	New

The sampled PC value is the PC of the graduating instruction in the current cycle. If the processor is stalled when the PC sample counter overflows, then the sampled PC is the PC of the next graduating instruction. The processor continues to sample the PC value even when it is in Debug mode.

Note that some of the smaller sample periods can be too few cycles with respect to the time needed to read out the sampled value. That is, it might take 41 clock ticks to read a sample, while the smallest sample period is 32, and the processor might overwrite the sample before it has been fully read out. Hence, the sample rate must be set to some appropriately large value to get a reasonable reading of the sampled PC values.

5.2.1 PC Sampling in Wait State

Note that the processor samples PC even when it is asleep, that is, in a WAIT state. This permits an analysis of the amount of time spent by a processor in WAIT state which may be used for example to revert to a low power mode during the non-execution phase of a real-time application. But counting cycles to update the PC sample value is a waste of power. Hence, when in a WAIT state, the processor must simply switch the New bit to 1 each time it is set to 0 by the probe hardware. Hence, the external agent or probe reading the PC value will detect a WAIT instruction for as long as the processor remains in the WAIT state. When the processor leaves the WAIT state, then counting is resumed as before.

5.2.2 PC Sampling a MT Processor

In a multi-VPE implementation of a processor with MIPS MT, each VPE has its own TAP controller and will independently sample the PC of the instructions executing in that VPE of the processor. In the context of a VPE, PC sampling cannot be enabled for a VPE until that VPE is enabled. If there are no active TCs on a given VPE then no new PC samples at available at the TAP controller PCsample register even if PCSe bit is 1. In general, in a processor with MT, it makes sense to leave the PCSe bit disabled until the system has booted and all VPEs are enabled and up and running before setting PCSe bit to 1. Otherwise, the PC sampling counter will continue to run and consume power even if there is nothing happening on a VPE and is it disabled in one way or another.

Chapter 6

EJTAG Processor Core Extensions

This chapter describes the behavior for processors that support EJTAG. It contains the following sections:

- Section 6.1 "Overview"
- Section 6.2 "Debug Mode Execution"
- Section 6.3 "Debug Exceptions"
- Section 6.4 "Debug Mode Exceptions"
- Section 6.5 "Interrupts and NMIs"
- Section 6.6 "Reset and Soft Reset of Processor"
- Section 6.8 "EJTAG Instructions"
- Section 6.7 "EJTAG Coprocessor 0 Registers"

6.1 Overview

The extensions for EJTAG provide the following major features:

- Debug Mode, associated exceptions and dedicated debug vector
- Instruction set extensions: SDBBP (Software Debug Breakpoint) and DERET (Debug Exception Return)
- CP0 registers: Debug, DEPC and DESAVE
- Memory-mapped debug segment (dseg) (optional)
- Interrupt and NMI control from Debug Control Register (DCR) (optional)
- Single step (optional)
- Debug interrupt request signal (optional)

Note that some of the features are optional.

The general description in this chapter covers MIPS32 and MIPS64 processors, implying an R4k-like privileged environment. Differences for processors with R3k privileged environments are described in Appendix A, "Differences for R3k Privileged Environments" on page 161.

6.2 Debug Mode Execution

Debug Mode is entered only through a debug exception. It is exited as a result of either execution of a DERET instruction or application of a reset or soft reset.

When the processor is operating in Debug Mode it has access to the same resources, instructions, and CP0 registers as in Kernel Mode. Restrictions on Kernel Mode access (non-zero coprocessor references, access to extended addressing controlled by UX, SX, KX, etc.) apply equally to Debug Mode, but Debug Mode provides some additional capabilities as described in this chapter.

Other processor modes (Kernel Mode, Supervisor Mode, User Mode) are collectively considered as Non-Debug Mode. Debug software can determine if the processor is in Non-Debug Mode or Debug Mode through the DM bit in the Debug register.

A debug exception in a processor implementing the MIPS MT ASE will cause all other TCs (Thread Contexts) in the processor, except the one executing the exception handler, to be suspended from concurrent execution until the DERET. Debug mode execution takes priority over all other TC scheduling rules in MIPS MT. A TC which is otherwise not permitted to issue instructions, due to a Halted, non-Activated (see the MIPS MT specification) or OffLine state (see Section 6.7.1 on page 104) may still be used to service a debug exception.

When a MIPS MT processor is operating in Debug Mode, it has access to the same resources and capabilities as if the VPE in Debug Mode had the MVP bit of the VPEConf0 register set, allowing access to all VPEs of the processor.

The ability of an OffLine MIPS MT TC to execute in Debug mode makes it possible for EJTAG-based debuggers to allow other TCs and/or other VPEs to continue executing while a particular TC has been stopped for debugging. The Debug exception handler can cause the TC to put itself, and/or other TCs, in an OffLine state, then execute a DERET. On exiting Debug mode, the processor will resume normal scheduling of “on-line” TCs, but the OffLine ones will remain frozen until released by, e.g. service of a subsequent DINT Debug exception.

It is not a requirement in EJTAG, but it is left as an implementation option in multiprocessor/multicore systems whether or not a global debug state is defined and can be set by the debugger to suspend other processors when one of the processors in a multi-core system encounters debug exception. Similarly, implementation can also trigger re-starting of other processors when the one in debug mode executes a DERET. See Appendix <TBD> for a description of this mechanism.

6.2.1 Debug Mode Instruction Set

The full native ISA of the processor is accessible in Debug Mode.

Coprocessor loads and stores to the dseg segment are not supported. The operation of the processor is UNDEFINED if a coprocessor load or store to dseg is executed in Debug Mode. Refer to Section 6.2.2 on page 82 for more information on the dseg address space.

6.2.2 Debug Mode Address Space

Debug Mode access to unmapped address space is identical to that of Kernel Mode. Mapped areas are accessible as in Kernel Mode, but only if a valid translation is immediately provided by the MMU.

This is because a memory access that would cause a TLB-type exception when tried from Kernel Mode, would, when tried from Debug Mode, cause re-entry into Debug Mode through an exception (see Section 6.4 on page 99). Memory accesses usually causing TLB-type exception are therefore not handled by the usual memory management routines if these memory accesses are made while in Debug Mode.

Updating and handling of cached areas is the same as that in Kernel Mode.

In addition, an optional uncached and unmapped debug segment dseg (EJTAG area) appears in the address range 0xFFFF FFFF FF20 0000 to 0xFFFF FFFF FF3F FFFF. The dseg segment thereby appears in the kseg part of the compatibility segment, and access to kseg is possible with the dseg segment provided as described in Section 6.2.2.1 on page 85 and Section 6.2.2.2 on page 86. Coprocessor loads and stores to the dseg segment are not allowed, as described in Section 6.2.1 on page 82.

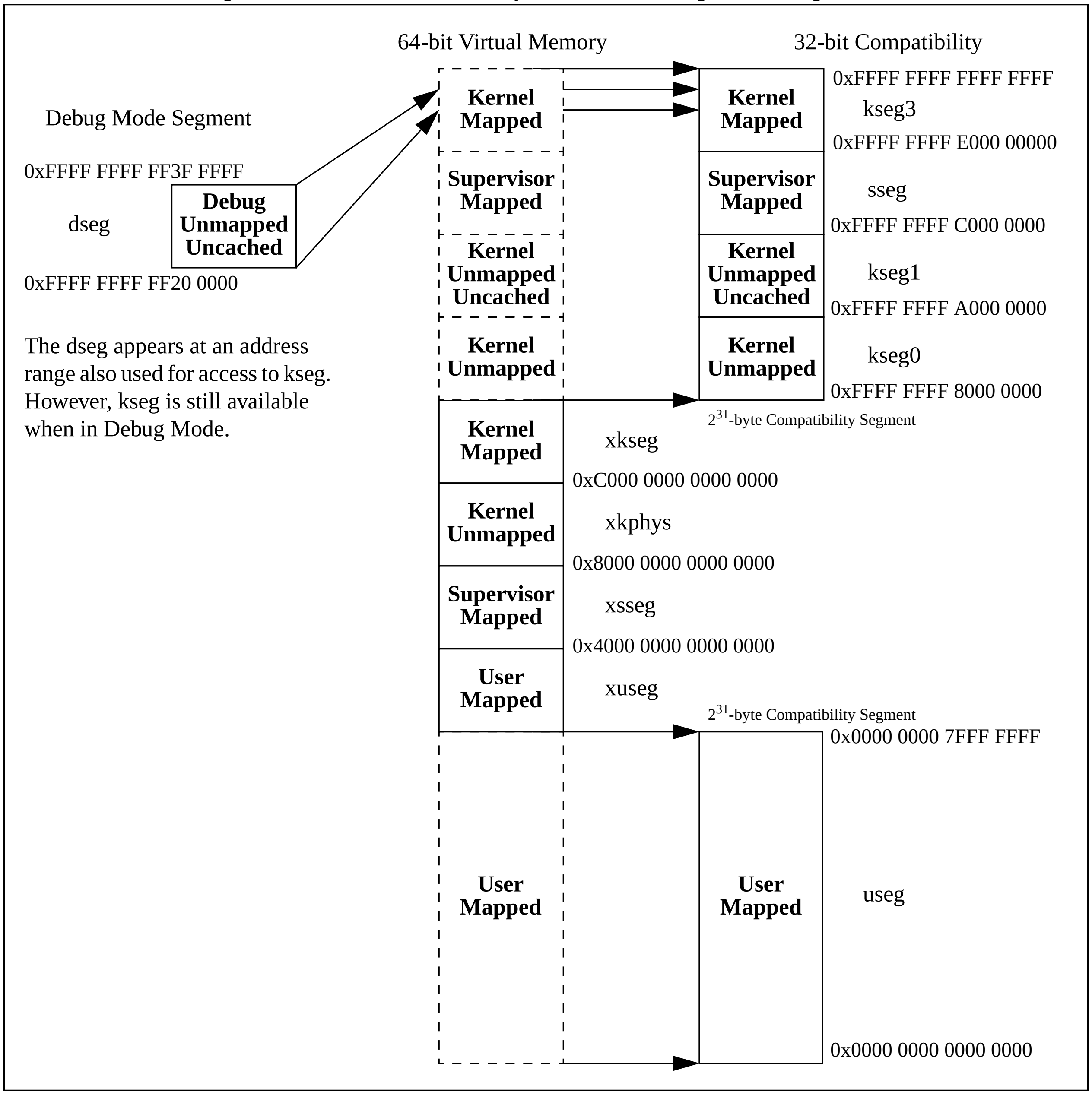
The dseg segment is implemented only if the Debug Control Register (DCR) is included in the implementation. Refer to Chapter 2, “Debug Control Register” on page 29 for more on the DCR. The implementation-dependent value of the NoDCR bit in the Debug register (see Section 6.7.1 on page 104) indicates the presence of the dseg segment as shown in Table 6.1. If the dseg segment is not present, then all transactions from the processor in Debug Mode go to the Kernel Mode address space. Debug software must check the $\text{Debug}_{\text{NoDCR}}$ bit before trying to access the dseg segment.

Table 6.1 Presence of the dseg Segment

NoDCR bit in Debug Register	dseg Presence
0	dseg Present
1	No dseg

Conditions for access to the dseg segment are described in Section 6.2.2.2 on page 86 and Section 6.2.2.1 on page 85. Figure 6.1 shows the layout of the virtual address space.

Figure 6.1 Virtual Address Spaces with Debug Mode Segments



The dseg segment is subdivided into dmseg (EJTAG memory) segment and the drseg (EJTAG registers) segment. The dmseg segment is used when the probe services the memory segment. The drseg segment is used when the memory-mapped debug registers are accessed. Table 6.2 shows the subdivision and attributes for the segments.

Table 6.2 Physical Address and Cache Attribute for dseg, dmseg and drseg

Segment Name	Subsegment Name	Virtual Address	Reference Address	Cache Attribute
dseg	dmseg	0xFFFF FFFF FF20 0000 to 0xFFFF FFFF FF2F FFFF	Because the dseg segment is serviced exclusively by the EJTAG features, there are no physical address per se. Instead the lower 21 bits of the virtual address select the appropriate reference in either EJTAG memory or registers. References are not mapped through the TLB, nor do the accesses appear on the external system memory interface.	Uncached
	drseg	0xFFFF FFFF FF30 0000 to 0xFFFF FFFF FF3F FFFF		

The SYNC instruction, followed by appropriate spacing (as described in Section 6.2.3.7 on page 88 and Section 6.2.4 on page 89) must be executed to ensure that an access to the dseg segment is committed (for example, after writing to the dseg segment and before leaving Debug Mode). This procedure ensures that locations in the dseg segment are fully updated for Non-Debug Mode, otherwise behavior of the processor is UNDEFINED.

6.2.2.1 Access to dmseg (EJTAG memory) Address Range

Table 6.3 shows the behavior of processor accesses in Debug Mode to the dmseg segment from 0xFFFF FFFF FF20 0000 to 0xFFFF FFFF FF2F FFFF.

Table 6.3 Access to dmseg Segment Address Range

NoDCR bit in Debug Register	Transaction	ProbEn bit in DCR register	LSNM bit in Debug Register	Access
1	x	(Not present)	0 (read-only)	Kernel Mode address space
0	Fetch	1	x	dmseg
		0	x	See comments below regarding behavior when ProbEn is 0
	Load/Store	1	0	dmseg
			1	Kernel Mode address space
		0	1	Kernel Mode address space
			0	See comments below regarding behavior when ProbEn is 0
‘x’ denotes don’t care				

From Table 6.3, when ProbEn equals 0 for dmseg segment accesses, debug software accessed the dmseg segment when the ProbEn bit was 0, indicating that there is no probe available to service the request. Debug software must read the state of the ProbEn bit in the DCR register before attempting to reference the dmseg segment. However, accessing the dmseg segment while ProbEn is 0 can occur because there is an inherent race between the debug software sampling the ProbEn bit as 1 and the probe clearing it to 0. The probe can therefore not assume that a reference to the dmseg segment never occurs if the ProbEn bit is dynamically cleared to 0. If debug software references the dmseg segment when ProbEn is 0, the reference hangs until it is satisfied by the probe.

There are no timing requirements with respect to transactions to the dmseg segment, which the probe services. Therefore a system watchdog must be disabled during dseg segment transactions, so accesses can take any amount of time without being terminated.

The protocol for accesses to the dmseg segment does not allow a transaction to be aborted once started, except by a reset or soft reset.

Transactions of all sizes are allowed to the dmseg segment.

Merging is allowed for accesses to the dmseg segment, whereby for example two byte accesses can be merged to one halfword access, and debug software is thus required to allow merging. However, merging must only occur for accesses which can be combined into legal processors accesses, since processor access can only indicate accesses which can occur due to a single load/store, thus not for example accesses to only first and last bytes of a word. The SYNC instruction, followed by appropriate spacing, (as described in Section 6.2.3.7 on page 88 and Section 6.2.4 on page 89) can be executed to ensure that earlier accesses to the dmseg segment are committed thus will not be merged with later accesses.

The processor can do speculative fetching from the dmseg segment whereby it can fetch doublewords even if an instruction that is not required in the execution flow is thereby fetched. For example if the DERET instruction is fetched as the first word of a doubleword, then the instruction in the second word is not executed. For details, refer to architecture description covering speculative fetching from uncached area in general.

If the TAP is not present in the implementation, then the operation of the processor is UNDEFINED if the dmseg segment is accessed.

6.2.2.2 Access to drseg (EJTAG Registers) Address Range

Table 6.4 shows the behavior of processor accesses in Debug Mode to the drseg segment from 0xFFFF FFFF FF30 0000 to 0xFFFF FFFF FF3F FFFF.

Table 6.4 Access to drseg Segment Address Range

NoDCR bit in Debug Register	Transaction	LSNM bit in Debug Register	Access
1	x	0 (read-only)	Kernel Mode address space
0	Fetch	x	Operation of the processor is UNDEFINED at fetch
	Load/Store	0	drseg segment (see comments below the table)
		1	Kernel Mode address space
‘x’ denotes don’t care			

Instruction fetches from the drseg segment are not allowed. The operation of the processor is UNDEFINED if the processor tries to fetch from the drseg segment.

When the NoDCR bit is 0 in the Debug register it indicates that the processor is allowed to access the entire drseg segment, therefore a response occurs to all transactions in the drseg segment.

The DCR register, at offset 0x0000 in the drseg segment, is always available if the dseg segment is present. Debug software is expected to read the DCR register to determine what other memory-mapped registers exist in the drseg segment. The value returned in response to a read of any unimplemented memory-mapped register is UNPREDICTABLE, and writes are ignored to any unimplemented register in the drseg segment.

The allowed transaction size is limited for the drseg segment. Only word size transactions are allowed for 32-bit processors, and only doubleword size transactions are allowed for 64-bit processors. Operation of the processor is UNDEFINED for other transaction sizes.

6.2.3 Debug Mode Handling of Processor Resources

Unless otherwise specified, the processor resources in Debug Mode are handled identically to those in Kernel Mode. Some identical cases are described in the following subsections for emphasis.

In addition, see the following related sections for more information:

- Section 6.4 "Debug Mode Exceptions" covering exception handling in Debug Mode.
- Section 6.5 "Interrupts and NMIs" for handling in both Debug and Non-Debug Modes.
- Section 6.6 "Reset and Soft Reset of Processor" for handling in both Debug and Non-Debug Modes.

6.2.3.1 Coprocessors

A Debug Mode Coprocessor Unusable exception is raised under the same conditions as for a Coprocessor Unusable exception in Kernel Mode (see Section 6.4.1 on page 99). Therefore Debug Mode software cannot reference Coprocessors 1 through 2 without first setting the respective enable in the Status register.

6.2.3.2 Random Register

For TLB-based MMU implementations, the Random register (CP0 register 1, select 0) optionally can be frozen in Debug Mode, whereby execution with and without debug exceptions are identical with respect to TLB exception handling.

If the values that the Random register provides cannot be identical in behavior to the case where debug exceptions do not occur, then freezing the Random register has no effect, because execution with and without debug exceptions will not be identical. Stalls when entering Debug Mode (for example, due to pending scheduled loads resolved at context save in the debug handler) can make it impossible in some implementations to ensure that the Random register will provide the same set of values when running with and without debug exceptions.

There is no bit to indicate or control if the Random register is frozen in Debug Mode, so the user must consult system documentation.

6.2.3.3 Count Register

The Count register (CP0 register 9) operation in Debug Mode depends on the state of the CountDM bit in the Debug register (see Section 6.7.1 on page 104). The Count Register has three possible configurations, depending on the implementation:

- Count register runs in Debug Mode the same as in Non-Debug Mode
- Count register is stopped in Debug Mode but is running in Non-Debug Mode
- The CountDM bit controls the Count register behavior in Debug Mode whereby it can be either running or stopped

Stopping of the Count register in Debug Mode is allowed in order to prevent generation of an interrupt at every return to Non-Debug Mode, if the debug handler takes so long to execute that the Count/Compare registers request an interrupt. In this case, system timing behavior might not be the same as if no debug exception occurred.

6.2.3.4 WatchLo/WatchHi Registers

The WatchLo/WatchHi registers (CP0 Registers 18 and 19) are inhibited from matching any instruction executed in Debug Mode.

6.2.3.5 CacheErr Register

The MIPS32 and MIPS64 architecture specifications state that operation of the CacheErr register is implementation dependent, so the CacheErr register handling described in the EJTAG Architecture is a recommendation only. Debug software can therefore not depend on the CacheErr register being implemented as recommended below.

The recommendation is that a CacheErr shadow register captures information presented when a cache error is indicated, and holds this information until a later update of the CacheErr register when a Cache Error exception occurs. The CacheErr shadow register is updated at “cache error indication AND (in Non-Debug Mode OR (in Debug Mode AND the IEXI bit is set))”. The CacheErr shadow register is not updated when in Debug Mode and the IEXI bit is cleared, but a cache error in this case only occurs due to an instruction executed in Debug Mode, if proper debug handler entry code is used. The CacheErr register is only updated at a Cache Error exception, thus not at a Debug Mode Cache Error exception.

If the CacheErr register value is to be correct for a cache error deferred through Debug Mode, then no cache errors may occur when in Debug Mode and the IEXI bit is set. The debug handler must therefore ensure the entry and exit code, executed with IEXI is set, can not cause cache errors, otherwise the CacheErr register contents presented to Non-Debug Mode is invalid.

6.2.3.6 Load Linked (LL/LLD) and Store Conditional (SC/SCD) Instruction Pair

A DERET instruction does not clear the LLbit (see “DERET” on page 117), neither does the occurrence of a debug exception. Loads and stores to uncacheable locations that do not match the physical address of the previous LL instruction do not affect the result of the SC instruction. The value of the LLbit is not directly visible by software.

6.2.3.7 SYNC and EHB Instruction Behavior

The SYNC instruction is used to request the hardware to commit certain operations before proceeding. For example, a SYNC is required to remove memory hazards on reference to the dseg segment. The EHB instruction ensures that status bits in the Debug register are fully updated before the debug handler accesses them and before Debug Mode is exited. Similarly, the SYNC instruction ensures that the hardware breakpoint registers in drseg memory address space are fully updated before the debug handler accesses them and before Debug Mode is exited. Cores implementing Release 2 of the architecture can use the EHB instruction (or Release 1 implementations can use SSNOP instructions combined with appropriate spacing), see Section 6.2.4 on page 89 to remove Coprocessor 0 (CP0) execution hazards.

The SYNC and EHB instructions must provide specific behavior as described in Table 6.5.

Table 6.5 SYNC and EHB Instruction References

Behavior	Section References
Commit accesses to the dseg segment	See Section 6.2.2 on page 82
Update the DDBLImpr and DDBSImpr bits in the Debug register	See Section 6.3.7 on page 95 and Section 6.7.1 on page 104
Update the BS bits in the IBS and DBS registers in drseg	See Section 3.4.2 on page 43

Table 6.5 SYNC and EHB Instruction References

Behavior	Section References
Update the IBusEP, DBusEP, CacheEP, and MCheckP bits in the Debug register	See Section 6.4.2 on page 100 and Section 6.7.1 on page 104

The SYNC instruction must be executed before leaving Debug Mode in order to commit all accesses to the dseg segment, for example to commit accesses to set up hardware breakpoints.

It may be required to remove hazards in relation to the SYNC instruction as described in Section 6.2.4 on page 89.

Other requirements of the SYNC instruction are described in the MIPS32 and MIPS64 Architecture specifications.

6.2.4 CP0 and dseg Segment Hazards

Because resources controlled via Coprocessor 0 and EJTAG memory and registers in the dseg segment affect the operation of various pipeline stages of the processor, manipulation of these resources may produce results that are not detectable by subsequent instructions for some number of execution cycles. When no hardware interlock exists between one instruction that causes an effect that is visible to a second instruction, a CP0 or dseg segment *hazard* exists.

In Release 1 of the MIPS32 and MIPS64 Architectures, hazards were relegated to implementation-dependent cycle-based solutions, primarily based on the SSNOP instruction. Since that time, it has become clear that this is an insufficient and error-prone practice that must be addressed with a firm compact between hardware and software. As such, new instructions have been added to Release 2 of the Architecture which act as explicit barriers that eliminate hazards. To the extent that it was possible to do so, the new instructions have been added in such a way that they are backward-compatible with existing MIPS processors.

6.2.4.1 Types of Hazards

In privileged software, there are two different types of hazards: execution hazards and instruction hazards. Both are defined below. In Table 6.6 below, the final column lists the “typical” spacing required in implementations of Release 1 of the Architecture to allow the consumer to eliminate the hazard. The “typical” value shown in these tables represent spacing that is in common use by operating systems today. An implementation of Release 1 of the Architecture which requires less spacing to clear the hazard (including one which has full hardware interlocking) should operate correctly with an operating system which uses this hazard table. An implementation of Release 1 of the Architecture which requires more spacing to clear the hazard incurs the burden of validating kernel code against the new hazard requirements.

Note that, for superscalar MIPS implementations, the number of instructions issued per cycle may be greater than one, and thus that the duration of the hazard in instructions may be greater than the duration in cycles. It is for this reason that MIPS Release 1 defines the SSNOP instruction to convert instruction issues to cycles in a superscalar design.

Execution Hazards

Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. Table 6.6 lists execution hazards related to EJTAG.

Table 6.6 Execution Hazards

Producer	→	Consumer	Hazard On	“Typical” Spacing (Cycles)
SYNC	→	DERET	dseg memory locations	2
SYNC	→	Load / Store	BS bits in the IBS and DBS registers in drseg	2
SYNC	→	MFC0 Debug	$Debug_{DDBSImpr}$, $Debug_{DDBLImpr}$, $Debug_{IBusEP}$, $Debug_{DBusEP}$, $Debug_{CacheEP}$, $Debug_{MCheckP}$	2
MTC0 DEPC	→	DERET	DEPC	2
MTC0 Debug	→	DERET	Debug	2
MTC0 Debug[LSNM]	→	Load / Store in dseg	Debug[LSNM]	3
MTC0 Debug[IEXI]	→	Instructions that can cause an imprecise exception	Debug[IEXI]	3

Dependencies from the SYNC instruction as producer takes effect since specific updates of the dseg segment and resolving of pending imprecise exception indications are triggered by the SYNC instruction. This is described in Section 6.2.3.7 on page 88.

Instruction Hazards

Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. There are no instruction hazards that are specific to EJTAG.

6.2.4.2 Hazard Clearing Instructions

Table 6.7 lists the instructions designed to eliminate hazards.

Table 6.7 Hazard Clearing Instructions

Mnemonic	Function
EHB	Clear execution hazard
JALR.HB	Clear both execution and instruction hazards
JR.HB	Clear both execution and instruction hazards
SSNOP	Superscalar No Operation
SYNCI	Synchronize caches after instruction stream write

6.2.4.3 Instruction Encoding

The EHB instruction is encoded using a variant of the NOP/SSNOP encoding. This encoding was chosen for compatibility with the Release 1 SSNOP instruction, such that existing software may be modified to be compatible with both Release 1 and Release 2 implementations. See the EHB instruction description for additional information.

The JALR.HB and JR.HB instructions are encoding using bit 10 of the *hint* field of the JALR and JR instructions. These encodings were chosen for compatibility with existing MIPS implementations, including many which pre-date the MIPS architecture. Because a pipeline flush clears hazards on most early implementations, the JALR.HB or JR.HB instructions can be included in existing software for backward and forward compatibility. See the JALR.HB and JR.HB instructions for additional information.

The SYNCI instruction is encoded using a new encoding of the REGIMM opcode. This encoding was chosen because it causes a Reserved Instruction exception on all Release 1 implementations. As such, kernel software running on processors that don't implement Release 2 can emulate the function using the CACHE instruction.

The SSNOP and EHB instructions are fully described in the MIPS32 and MIPS64 Architecture for Programmers, Volume II.

6.3 Debug Exceptions

This section describes issues related to debug exceptions. Debug exceptions bring the processor from Non-Debug Mode into Debug Mode. Implementations need only support those debug exceptions that are applicable to that implementation.

Exceptions can occur in Debug Mode, and these are denoted as debug mode exceptions. These exceptions are handled differently from exceptions that occur in Non-Debug Mode, which are described in Section 6.4 on page 99.

6.3.1 Debug Exception Priorities

Table 6.8 lists the exceptions that can occur in Non-Debug Mode in order of priority, from highest to lowest. The table also categorizes each exception with respect to type (debug or non-debug). Each debug exception has an associated status bit in the Debug register (indicated in the table in parentheses). Refer to Section 6.7.1 on page 104 for more information.

Table 6.8 Priority of Non-Debug and Debug Exceptions

Priority	Exception	Type of Exception
Highest	Reset	Non-debug
	Soft reset	
	Debug Single Step	Debug
	Debug Interrupt; by external signal (DINT), from EjtagBrk in TAP, or through use of EJTAG Boot.	
	Debug Data Break Load/Store Imprecise (DDBLImpr/DDBSImpr)	
	Nonmaskable Interrupt (NMI)	Non-debug
	Machine Check	
	Interrupt	
	Deferred Watch	
	Debug Instruction Break	Debug

Table 6.8 Priority of Non-Debug and Debug Exceptions (Continued)

Priority	Exception	Type of Exception
	Watch on instruction fetch	Non-debug
	Address error on instruction fetch	
	TLB refill on instruction Ifetch	
	TLB Invalid on instruction Ifetch	
	Cache error on instruction Ifetch	
	Bus error on instruction Ifetch	
	Debug Breakpoint; execution of SDBBP instruction	Debug
	Other execution-based exceptions	Non-debug
	Debug Data Break on Load/Store address match only or Debug Data Break on Store address+data value match	Debug
	Watch on data access	Non-debug
	Address error on data access	
	TLB Refill on data access	
	TLB Invalid on data access	
	TLB Modified on data access	
	Cache error on data access	
	Bus error on data access	
Lowest	Debug Data Break on Load address+data match	Debug

The specific implementation determines which exceptions can occur and the priority of asynchronous exceptions, such as interrupts.

6.3.2 Debug Exception Vector Location

The same debug exception vector location is used for all debug exceptions. The ProbTrap bit in the EJTAG Control Register (ECR) in the optional Test Access Port (TAP) determines the vector location. Starting with EJTAG ver 4.5x,

Table 6.9 Debug Exception Vector Location

ProbTrap bit in ECR register	Debug Exception Vector Address
0	0xFFFF FFFF BFC0 0480
1	0xFFFF FFFF FF20 0200 in dmseg

a new drseg register (DebugVectorAddr, at offset 0x00020) allows the debug exception vector to be relocated if the ProbTrap bit in the ECR register is 0. A new control bit is added in the EJTAG control register (ECR$_{RDVec}$) that enables or disables the relocation mechanism. Figure 6.2 shows the format of the DebugVectorAddr register; Table 6.10 describes the DebugVectorAddr register fields.

Figure 6.2 DebugVectorAddr Register Format

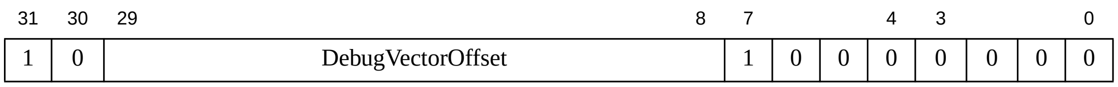


Table 6.10 DebugVectorAddr Register Field Descriptions

Fields		Description	Read / Write	Power-up State	Compliance
Name	Bits				
1	31,7	Ignored on write; returns one on read.	R	1	Required
DebugVectorOffset	29:8		R/W	Preset to 0x3FC004	Required
0	30,6-0	Ignored on write; returns zero on read.	R	0	Required

If the TAP is not implemented, then the debug exception vector location is as if ProbTrap is 0.

6.3.3 General Debug Exception Processing

All debug exceptions have the same basic processing flow:

- The DEPC register is loaded with the PC at which execution can be restarted, and the DBD bit is set to indicate whether the last debug exception occurred in a branch delay slot. The value loaded into the DEPC register is either the current PC (if the instruction is not in the delay slot of a branch) or the PC of the branch or jump (if the instruction is in the delay slot of a branch or jump).
- The DSS, DBp, DDBL, DDBS, DIB, DINT, DDBLImpr, and DDBSImpr bits in the Debug register are updated appropriately depending on the debug exception.
- DExcCode field in the Debug register is undefined.
- Halt and Doze bits in the Debug register are updated appropriately.
- IEXI bit is set to inhibit imprecise exceptions in the start of the debug handler.
- DM bit in the Debug register is set to 1.
- The processor begins fetching instructions from the debug exception vector.

The value loaded into the DEPC register represents the restart address from the debug exception and does not need to be modified by the debug exception handler software. Debug software need only look at the DBD bit in the Debug register if it wishes to identify the address of the instruction that actually caused a precise debug exception.

The DSS, DBp, DDBL, DDBS, DIB, DINT, DDBLImpr, and DDBSImpr bits in the Debug register indicate the occurrence of distinct debug exceptions, except when a Debug Data Break Load/Store Imprecise exception occurs (see Section 6.3.7 on page 95). Note that occurrence of an exception while in Debug mode will clear these bits. The handler can thereby determine whether a debug exception or an exception in Debug Mode occurred.

Also note that multiple cause bits may be set, but the priority of the debug exception or interrupt dictates the order in which they are handled. For example, since DSS is the highest priority Debug exception, if it occurs, it will always be taken first. Then, after it DERETS, other debug exceptions can be taken. For example, assume that the processor is in single-step mode in a branch delay slot, and waiting to go past the delay slot to enter the DSS exception. At the branch delay slot, it could get a DINT or other lower priority Debug exception. In this case, it would not take the lower exception, but enter Debug Mode past the delay slot. The entry into Debug Mode will clear the DINT. It would process the single-step exception and DERET to normal non-debug mode. Note that in practice, not many cores set multiple cause bits in the Debug register since the highest priority debug exception is taken, and the others are cleared on entry to Debug Mode as already specified.

No other CP0 registers or fields are changed due to the debug exception, thus no additional state is saved.

The overall exception processing flow happens in hardware before setting PC to point to the debug exception vector is shown below:

Operation:

```
if (InstructionInBranchDelaySlot) then
    DEPC ← BranchInstructionPC
    $Debug_{DBD}$ ← 1
else
    DEPC ← PC
    $Debug_{DBD}$ ← 0
endif
$Debug_{DSS, DBp, DDBL, DDBS, DIB, DINT, DDBLImpr and DDBSImpr}$ ← DebugExceptionType
$Debug_{DExcCode}$ ← UNPREDICTABLE
$Debug_{Halt}$ ← HaltStatusAtDebugException
$Debug_{Doze}$ ← DozeStatusAtDebugException
$Debug_{IEXI}$ ← 1
$Debug_{DM}$ ← 1
if $ECR_{ProbTrap}$ = 1 then
    PC ← 0xFFFF FFFF FF20 0200
else
    PC ← 0xFFFF FFFF BFC0 0480
endif
```

6.3.4 Debug Breakpoint Exception

A Debug Breakpoint exception occurs when an SDBBP instruction is executed. The contents of the DEPC register and the DBD bit in the Debug register indicate that the SDBBP instruction caused the debug exception.

Debug Register Debug Status Bit Set

DBp

Additional State Saved

None

Entry Vector Used

Debug exception vector

6.3.5 Debug Instruction Break Exception

A Debug Instruction Break exception occurs when an instruction hardware breakpoint matches an executed instruction. The DEPC register and DBD bit in the Debug register indicate the instruction that caused the instruction hardware breakpoint match. This exception can only occur if instruction hardware breakpoints are implemented (see Chapter 3, “Hardware Breakpoints” on page 33).

Debug Register Debug Status Bit Set

DIB

Additional State Saved

None

Entry Vector Used

Debug exception vector

6.3.6 Debug Data Break Load/Store Exception

A Debug Data Break Load/Store exception occurs when a data hardware breakpoint matches the load/store address of an executed load/store instruction. The DEPC register and DBD bit in the Debug register indicate the load/store instruction that caused the data hardware breakpoint to match, as this is a precise debug exception. The load/store instruction that caused the debug exception has not completed (it has not updated the destination register or memory location), and the instruction therefore is executed on return from the debug handler. This exception can only occur if data hardware breakpoints with precise data breaks are implemented (see Chapter 3, “Hardware Breakpoints” on page 33).

Debug Register Debug Status Bit Set

DDBL for a load instruction or DDBS for a store instruction

Additional State Saved

None

Entry Vector Used

Debug exception vector

6.3.7 Debug Data Break Load/Store Imprecise Exception

A Debug Data Break Load/Store Imprecise exception occurs when a data hardware breakpoint matches a load/store access of an executed load/store instruction, if it is not possible to take a precise debug exception on the instruction. This case occurs when a data hardware breakpoint was set up with a value compare, and a load access did not return data until after the load instruction had left the pipeline as for non-blocking loads. The DEPC register and the DBD bit in the Debug register indicate an instruction later in the execution flow instead of the load/store instruction that caused the data hardware breakpoint to match. The DDBLImpr/DDBSImpr bits in the Debug register indicate that a Debug Data Break Load/Store Imprecise exception occurred. The instruction that caused the Debug Data Break Load/Store Imprecise exception will have completed. It updates its destination register, and is not executed on return from the debug handler. This exception can only occur if data hardware breakpoints with imprecise data breakpoints are implemented (see Chapter 3, “Hardware Breakpoints” on page 33).

Imprecise debug exceptions from data hardware breakpoints are indicated together with another debug exception if the load/store transaction that made the data hardware breakpoint match did not complete until after another debug exception occurred. In this case, the other debug exception was the cause of entering Debug Mode, so the DEPC register and the DBD bit in Debug register point to this instruction. DDBLImpr/DDBSImpr are set concurrently with the status bit for that debug exception.

The SYNC followed by appropriate spacing and the EHB instruction, (as described in Section 6.2.3.7 on page 88 and Section 6.2.4 on page 89) must be executed in Debug Mode before the DDBLImpr and DDBSImpr bits in the Debug register and the BS bits for the data hardware breakpoint are respectively read in order to ensure that all imprecise breaks are resolved and the bits are fully updated. A match of the data hardware breakpoint is indicated in DDBLImpr/DDBSImpr so the debug handler can handle this together with the debug exception.

This scheme ensures that all breakpoints matching due to code executed before the debug exception are indicated by the DDBLImpr, DDBSImpr, and BS bits for the following debug handler. Matches are neither queued nor do they cause debug exceptions at a later point. A debug exception occurring later than the debug exception handler is therefore caused by code executed in Non-Debug Mode after the debug exception handler.

Debug Register Debug Status Bit Set

DDBLImpr for a load instruction or DDBSImpr for a store instruction

Additional State Saved

None

Entry Vector Used

Debug exception vector

6.3.8 Debug Single Step Exception

When single-step mode is enabled, a Debug Single Step exception occurs each time the processor has taken a single execution step in Non-Debug Mode. An execution step is a single instruction, or an instruction pair consisting of a jump/branch instruction and the instruction in the associated delay slot. The *SSt* bit in the Debug register enables Debug Single Step exceptions. They are disabled on the first execution step after a DERET.

The DEPC register points to the instruction on which the Debug Single Step exception occurred, which is also the next instruction to execute when returning from Debug Mode. The debug software can examine the system state before this instruction is executed. Thus the DEPC will not point to the instruction(s) that have just executed in the execution step, but rather the instruction following the execution step. The Debug Single Step exception never occurs on an instruction in a jump/branch delay slot, because the jump/branch and the instruction in the delay slot are always executed in one execution step; thus the DBD bit in the Debug register is never set for a Debug Single Step exception.

Exceptions occurring on the instruction(s) in the execution step are taken regardless, so if a non-debug exception occurs (other than reset or soft reset), a Debug Single Step exception is taken on the first instruction in the non-debug exception handler. The non-debug exception occurs during the execution step, and the instruction(s) that received a non-debug exception counts as the execution step.

Debug exceptions are unaffected by single-step mode; returning to an SDBBP instruction with single step enabled causes a Debug Breakpoint exception with the DEPC register pointing to the SDBBP instruction. Also, returning to an instruction (not jump/branch) just before the SDBBP instruction causes a Debug Single Step exception with the DEPC register pointing to the SDBBP instruction.

To ensure proper functionality of single-step execution, the Debug Single Step exception has priority over all exceptions, except resets and soft resets.

Debug Single Step exception is only possible when the NoSSt bit in the Debug register is 0 (see Section 6.7.1 on page 104).

In an core that implements the MIPS MT ASE, the *SSt* bit is instantiated per TC. If the *SSt* bit of the TC is set, a Debug exception will be taken by that TC after any non-Debug mode instruction is executed. Other TCs with *SSt* cleared are scheduled and issue instructions normally according to the scheduling policy in force. Global single-step operation of a VPE can be achieved by setting *SSt* for all TCs for the specified VPE.

When the single-step exception bit is set for multiple TCs, then the preferred behavior applies it to each TC independently and independent of the scheduling policy. This has implications for the software observable instruction execution completion order. Three examples are shown in Figure 6.3, Figure 6.4, and Figure 6.5. In Figure 6.3 there are two threads TC0 and TC1, and thread TC0 has its *SSt* bit set but thread TC1 does not have its *SSt* bit set. In Figure 6.4, there are two threads and both their *SSt* bits are set. In Figure 6.5, there are four threads, and two threads have their *SSt* bits set and the other two do not. The figures show the observed instruction completion order for each of the cases. The notation used is TC#.Instn#.

Debug Register Debug Status Bit Set

DSS

Additional State Saved

None

Entry Vector Used

Debug exception vector

Figure 6.3 Example 1: Single-stepping One Thread TC0 with Non-single-Stepping Thread TC1

```
0.0 - DSS
0.x - dexc
0.x - DERET
1.0 - completes
0.0 - completes
0.1 - DSS
0.x - dexc
0.x - DERET
1.1 - completes
0.1 - completes
...
```

Figure 6.4 Example 2: Single-stepping Two Threads TC0 and TC1

```
0.0 - DSS
0.x - dexc handler
0.x - DERET
1.0 - DSS
1.x - dexc handler
1.x - DERET
0.0 - completes
1.0 - completes
0.1 - DSS
0.x - dexc handler
0.x - DERET
...
```

Figure 6.5 Example 3: Single-stepping Two Threads TC0 and TC1 with Other Threads TC2 and TC3

```
0.0 - DSS
0.x - dexc handler
0.x - DERET
1.0 - completes
2.0 - DSS
2.x - dexc handler
2.x - DERET
3.0 - completes
0.0 - completes
1.1 - completes
2.0 - completes
3.1 - completes
0.1 - DSS
0.x - dexc handler
0.x - DERET
1.2 - completes
...
```

6.3.9 Debug Interrupt Exception

The Debug Interrupt exception is an asynchronous debug exception that is taken as soon as possible, but with no specific relation to the executed instructions. The DEPC register and the DBD bit in the Debug register reference the instruction at which execution can be resumed after Debug Interrupt exception service.

Debug interrupt requests are ignored when the processor is in Debug Mode, and pending requests are cleared when the processor takes any debug exception, including debug exceptions other than Debug Interrupt exceptions.

A debug interrupt restarts the pipeline if stopped by a WAIT instruction and the processor clock is restarted if it was stopped due to a low-power mode.

Debug Register Debug Status Bit Set

DINT

Additional State Saved

None

Entry Vector Used

Debug exception vector

The possible sources for debug interrupts depend on the implementation. The following sources can cause Debug Interrupt exceptions:

- The DINT signal from the probe

  The optional DINT signal from the probe can request a debug interrupt on a low (0) to high (1) transition. The DINTsup bit in the Implementation register in the Test Access Port (TAP) indicates whether the DINT signal from the probe to the target processor is implemented (see Section 7.5.2 on page 128). The timing requirements for the DINT signal are shown in Section 9.2.2 on page 154.

The DINT signal can be synchronized to the processor clock domain before edge detection while still observing the required timing of the DINT signal. If the CPU clock speed or clocking scheme is such that the required timing does not leave enough time for synchronization or clock wake-up, then the DINT pulse is extended by the target system in the processor.

The EjtagBrk bit in the EJTAG Control register provides similar functionality similar to DINT from the probe, but with higher latency.

- The EjtagBrk Bit in the EJTAG Control Register

  The EjtagBrk bit in the EJTAG Control register requests a Debug Interrupt exception when set (see Section 7.5.5 on page 133).

- A debug boot by EJTAGBOOT

  The EJTAGBOOT feature allows a debug interrupt to be requested immediately after a reset or soft reset has occurred, and before the first instruction is fetched from the reset exception vector (see Section 6.6.1 on page 103 and Section 7.4.2 on page 125).

- An implementation-specific debug interrupt signal to the processor

  Through the availability of an optional debug interrupt request signal to the processor system, an external device can request a Debug Interrupt exception, for example, when a signal goes from deasserted to asserted.

6.4 Debug Mode Exceptions

The handling of exceptions generated in Debug Mode, other than through resets and soft resets, differs from those exceptions generated in Non-Debug Mode in that only the Debug and DEPC registers are updated. All other CP0 registers are unchanged by an exception taken in Debug Mode. The exception vector is equal to the debug exception vector (see Section 6.3.2 on page 92), and the processor stays in Debug Mode.

Reset and soft reset are handled as when occurring in Non-Debug Mode (see Section 6.6 on page 103).

6.4.1 Exceptions Taken in Debug Mode

Only some Non-Debug Mode exception events cause exceptions while in Debug Mode. Remaining events are blocked. Exceptions occurring in Debug Mode have the same relative priorities as the Non-Debug Mode exceptions for the same exception event. These exceptions are called Debug Mode <Non-Debug Mode exception name>. For example, a Debug Mode Breakpoint exception is caused by execution of a BREAK instruction in Debug Mode, and a Debug Mode Address Error exception is caused by an address error due to an instruction executed in Debug Mode.

Table 6.11 lists all the Debug Mode exceptions with their corresponding non-debug exception event names, priorities, and handling.

Table 6.11 Exception Handling in Debug Mode

Priority	Event in Debug Mode	Debug Mode Handling
Highest	Reset	Reset and soft reset handled as for Non-Debug Mode, see Section 6.6 on page 103.
	Soft reset	

Table 6.11 Exception Handling in Debug Mode (Continued)

Priority	Event in Debug Mode	Debug Mode Handling
	Debug Single Step	Blocked
	Debug Interrupt	
	Debug Data Break Load/Store Imprecise	
	NMI	
	Machine Check	Re-enter Debug Mode
	Interrupt	Blocked
	Deferred Watch	
	Debug Instruction Break, DIB	
	Watch on instruction fetch	
	Address error on instruction fetch	Re-enter Debug Mode
	TLB refill on instruction Ifetch	
	TLB Invalid on instruction Ifetch	
	Cache error on instruction Ifetch	
	Bus error on instruction Ifetch	
	Debug Breakpoint; execution of SDBBP instruction	Re-enter Debug Mode as for execution of the BREAK instruction
	Other execution-based exceptions	Re-enter Debug Mode
	Debug Data Break Load/Store address match only or Debug Data Break Store address+data value match	Blocked
	Watch on data access	
	Address error on data access	Re-enter Debug Mode
	TLB Refill on data access	
	TLB Invalid on data access	
	TLB Modified on data access	
	Cache error on data access	
	Bus error on data access	
Lowest	Debug Data Break on Load address+data match	Blocked

The specific implementation determines which exceptions can occur. Exceptions that are blocked in Debug Mode are simply ignored, not causing updates in any state.

Handling of the exceptions causing Debug Mode re-enter are described below.

6.4.2 Exceptions on Imprecise Errors

Exceptions on imprecise errors are possible in Debug Mode due to a bus error on an instruction fetch or data access, cache error, or machine check.

The IEXI bit in the Debug register blocks imprecise error exceptions on entry or re-entry into Debug Mode. They can be re-enabled by the debug exception handler once sufficient context has been saved to allow a safe re-entry into Debug Mode and the debug handler.

Pending exceptions due to instruction fetch bus errors, data access bus errors, cache errors, and machine checks are indicated and controlled by the IBusEP, DBusEP, CacheEP and MCheckP bit in the Debug register.

The SYNC instruction, followed by appropriate spacing and the EHB instruction, (as described in Section 6.2.3.7 on page 88 and Section 6.2.4 on page 89) must be executed in Debug Mode before the IBusEP, DBusEP, CacheEP, and MCheckP bits are read in order to ensure that all pending causes for imprecise errors are resolved and all bits are fully updated.

Those bits required to handle the possible imprecise errors in an implementation are implemented as R/W, otherwise they are read only.

6.4.3 Debug Mode Exception Processing

All exceptions that are allowed in Debug Mode (except for reset and soft reset) have the same basic processing flow:

- The DEPC register is loaded with the PC at which execution will be restarted and the DBD bit is set appropriately in the Debug register. The value loaded into the DEPC register is either the current PC (if the instruction is not in the delay slot of a branch or jump) or the PC of the branch or jump if the instruction is in the delay slot of a branch or jump).
- The DSS, DBp, DDBL, DDBS, DIB, DINT, DDBLImpr, and DDBSImpr bits in the Debug register are all cleared to differentiate from debug exceptions where at least one of the bits are set.
- The DExcCode field in the Debug register is updated to indicate the type of exception that occurred.
- The Halt and Doze bits in the Debug register are UNPREDICTABLE.
- The IEXI bit is set to inhibit imprecise exceptions at the start of the debug handler.
- The DM bit in the Debug register is unchanged, leaving the processor in Debug Mode.
- The processor is started at the debug exception vector, specified in Section 6.3.2 on page 92.

The value loaded into the DEPC register represents the restart address for the exception; typically debug software does not need to modify this value at the location of the debug exception. Debug software need not look at the DBD bit in the Debug register unless it wishes to identify the address of the instruction that actually caused the exception in Debug Mode.

It is the responsibility of the debug handler to save the contents of the Debug, DEPC, and DESAVE registers before nested entries into the handler at the debug exception vector can occur. The handler returns to the debug exception handler by a jump instruction, not a DERET, in order to keep the processor in Debug Mode.

The cause of the exception in Debug Mode is indicated through the DExcCode field in the Debug register, and the same codes are used for the exceptions as those for the ExcCode field in the Cause register when the exceptions with the same names occur in Non-Debug Mode, with addition of the code 30 (decimal) with the mnemonic CacheErr for cache errors.

No other CP0 registers or fields are changed due to the exception in Debug Mode. For example, if the implementation supports setting of the TS bit in the CP0 Status register on the detection of a match on multiple TLB entries before a machine check exception, then the write of this TS bit should be suppressed when the machine check exception occurs in Debug mode.

The overall processing flow for exceptions in Debug Mode is shown below:

Operation:

```
if (InstructionInBranchDelaySlot) then
    DEPC ← BranchInstructionPC
    $Debug_{DBD}$ ← 1
else
    DEPC ← PC
    $Debug_{DBD}$ ← 0
endif
$Debug_{DSS,\ DBp,\ DDBL,\ DDBS,\ DIB,\ DINT,\ DDBLImpr\ and\ DDBSImpr}$ ← 0
$Debug_{DExcCode}$ ← DebugExceptionType
$Debug_{Halt}$ ← UNPREDICTABLE
$Debug_{Doze}$ ← UNPREDICTABLE
$Debug_{IEXI}$ ← 1
if $ECR_{ProbTrap}$ = 1 then
    PC ← 0xFFFF FFFF FF20 0200
else
    PC ← 0xFFFF FFFF BFC0 0480
endif
```

6.5 Interrupts and NMIs

Interrupts and NMIs are handled for EJTAG-compliant processors as described in the following subsections.

6.5.1 Interrupts

Interrupts are requested through either asserted external hardware signals or internal software-controllable bits. Interrupt exceptions are disabled when any of the following conditions are true:

- The processor is operating in Debug Mode
- The Interrupt Enable (IntE) bit in the Debug Control Register (DCR) is cleared (see Section Table 2.1 “DCR Register Field Descriptions”)
- A non-EJTAG related mechanism disables the interrupt exception

A pending interrupt is indicated through the Cause register, even if Interrupt exceptions are disabled.

6.5.2 NMIs

An NMI is requested on the asserting edge of the NMI signal to the processor, and an internal indicator holds the NMI request until the NMI exception is actually taken.

NMI exceptions are disabled when either of the following is true:

- The Processor is operating in Debug Mode
- The NMI Enable (NMIE) bit in the Debug Control Register (DCR) is cleared, see Section Table 2.1 “DCR Register Field Descriptions”

If an asserting edge on the NMI signal to the processor is detected while NMI exception is disabled, then the NMI request is held pending and is deferred until NMI exceptions are no longer disabled.

A pending NMI is indicated in the NMIpend bit in the DCR even if NMI exceptions are disabled.

6.6 Reset and Soft Reset of Processor

This section covers the handling of issues with respect to resets and soft resets. For EJTAG features, there are no difference between a reset and a soft reset occurring to the processor; they behave identically in both Debug Mode and Non-Debug Mode. References to reset in the following therefore refers to both reset (hard reset) and soft reset.

6.6.1 EJTAGBOOT Feature

The EJTAGBOOT feature allows a debug interrupt to be requested as a result of a reset instead of the regular reset exception. That is, instructions from the Debug Interrupt exception are fetched before any of the instructions from the reset exception vector are fetched.

The location of the debug exception handler is controlled by the ProbTrap bit in the TAP Control register. When this bit is set, the instructions for the debug exception handler are provided by the probe through the dmseg segment, taking care of a situation where the normal memory system does not work properly.

Control and details of EJTAGBOOT are described in Section 7.4.2 on page 125 and Table 7.9 describes the ProbTrap bit in the EJTAG Control register.

6.6.2 Reset from Probe

While asserted, the RST* signal from the probe is required to generate a reset or soft reset to the system. The SRstE bit in the Debug Control Register does not mask this source. See Section 9.1.3 on page 151 for more information.

6.6.3 Processor Reset by Probe through Test Access Port

The PrRst bit in the EJTAG Control register can optionally cause a reset depending on the implementation. If a reset occurs, then all parts of the system are reset, because partial resets are not allowed.

6.6.4 Reset Occurred Indication through Test Access Port

The Rocc bit in the EJTAG Control register is set at both reset and soft reset in order to indicate the event to the probe.

Refer to Section 7.5.5 on page 133 for more information on the EJTAG Control Register.

6.6.5 Soft Reset Enable

The optional Soft Reset Enable (SRstE) bit in the Debug Control Register (DCR) can mask the soft reset signal outside the processor. Because SRstE masks the soft reset signal before it arrives at the processor, there is no masking of soft reset within the processor itself.

6.6.6 Reset of Other Debug Features

The operation of processor resets and soft resets also apply to resets of the following:

- Debug Control Register (DCR), see Chapter 2, “Debug Control Register” on page 29
- Hardware Breakpoint, see Chapter 3, “Hardware Breakpoints” on page 33

- Test Access Port (TAP) EJTAG Control Register, see Chapter 7, “EJTAG Test Access Port” on page 119

6.7 EJTAG Coprocessor 0 Registers

The Coprocessor 0 registers for EJTAG are shown in Table 6.12. Each register is described in more detail in the following subsections.

Table 6.12 Coprocessor 0 Registers for EJTAG

Register Number	Sel	Register Name	Function	Reference	Compliance Level
23	0	Debug	Debug indications and controls for the processor.	See Section 6.7.1 on page 104	Required
24	0	DEPC	Program counter at last debug exception or exception in Debug Mode.	See Section 6.7.3 on page 113	Required
31	0	DESAVE	Debug exception save register.	See Section 6.7.4 on page 114	Required

The CP0 instructions MTC0, MFC0, DMTC0, and DMFC0 work with the three EJTAG CP0 registers as per the MIPS32 and MIPS64 Architecture specifications.

Operation of the processor is UNDEFINED if the Debug, DEPC, or DESAVE registers are written from Non-Debug Mode. The value of the Debug, DEPC, or DESAVE registers is UNPREDICTABLE when read from Non-Debug Mode, unless otherwise explicitly stated in the individual register description. However, for test purposes, the implementations can allow writes to and reads from the registers from Non-Debug Mode.

To avoid pipeline hazards, there must be an appropriate spacing, refer to Section 6.2.4 on page 89, between the update of the Debug and DEPC registers by MTC0/DMTC0 and use of the new value. This applies for example to modification of the LSNM bit of the Debug register and a load/store affected by that bit.

In a processor implementing the MIPS MT ASE, each of the Coprocessor 0 EJTAG registers described above is instantiated per VPE. The exception is the *SSt* and *OffLine* bits in the Debug register which is instantiated per-TC.

6.7.1 Debug Register (CP0 Register 23, Select 0)

Compliance Level: Required for EJTAG debug support.

The Debug register contains the cause of the most recent debug exception and exception in Debug Mode. It also controls single stepping. This register indicates low-power and clock states on debug exceptions, debug resources, and other internal states.

Only the DM bit and the EJTAGver field are valid when read from the Debug register in Non-Debug Mode; the value of all other bits and fields is UNPREDICTABLE.

The following bits and fields are only updated on debug exceptions and/or exceptions in Debug Mode:

- DSS, DBp, DDBL, DDBS, DIB, DINT, DIBImpr, DDBLImpr, and DDBSImpr are updated on both debug exceptions and on exceptions in Debug Modes

- DExcCode is updated on exceptions in Debug Mode, and is undefined after a debug exception

- Halt and Doze are updated on a debug exception, and are undefined after an exception in Debug Mode. In the situation where the processor is awakened from sleep or doze state by a hardware interrupt or other external event, and a debug exception is taken instead (for example, if single-stepping a WAIT instruction), the state of the Halt and Doze bits should be as if the hardware interrupt had not occurred. That is, these bits should indicate that the state of the processor was in Halt or Doze respectively before the exception, ignoring that the interrupt time might be between halt/doze and the debug exception.

- DBD is updated on both debug and on exceptions in Debug Modes

The SYNC instruction, followed by appropriate spacing and the EHB instruction, (as described in Section 6.2.3.7 on page 88 and Section 6.2.4 on page 89) must be executed to ensure that the DDBLImpr, DDBSImpr, IBusEP, DBusEP, CacheEP, and MCheckP bits are fully updated. This instruction sequence must be used both in the beginning of the debug handler before pending imprecise errors are detected from Non-Debug Mode, and at the end of the debug handler before pending imprecise errors are detected from Debug Mode. The IEXI bit controls enable/disable of imprecise error exceptions.

Figure 6.6 shows the format of the Debug register; Table 6.13 describes the Debug register fields.

Figure 6.6 Debug Register Format

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
32/64-bit Processor	DBD	DM	No DCR	LSNM	Doze	Halt	Count DM	IBus EP	M CheckP	Cach eEP	DBus EP	IEXI	DDB S Impr	DDB L Impr	EJTAGver [2:1]	
	15	14				10	9	8	7	6	5	4	3	2	1	0
	EJTA Gver [0]	DExcCode					NoSSt	SSt	OffLine	DIBI mpr	DINT	DIB	DDB S	DDB L	DBp	DSS

Table 6.13 Debug Register Field Descriptions

Fields		Description	Read/Write	Reset State	Compliance
Name	**Bits**				
DBD	31	Indicates whether the last debug exception or exception in Debug Mode occurred in a branch or jump delay slot: **Encoding** / **Meaning** 0 — Not in delay slot 1 — In delay slot	R	Undefined	Required
DM	30	Indicates that the processor is operating in Debug Mode: **Encoding** / **Meaning** 0 — Processor is operating in Non-Debug Mode 1 — Processor is operating in Debug Mode	R	0	Required

Table 6.13 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
NoDCR	29	Indicates whether the dseg segment is present: Encoding / Meaning 0: dseg segment is present 1: dseg present is not present	R	Preset	Required
LSNM	28	Controls access of loads/stores between the dseg segment and remaining memory when the dseg segment is present: Encoding / Meaning 0: Loads/stores in the dseg segment address range go to the dseg segment 1: Loads/stores in dseg segment address range go to system memory Further description in Section 6.2.2 on page 82. If DCR is not implemented, this bit is read-only (R) and reads as zero.	R/W	0	Required if the dseg segment is present, otherwise not implemented. See bit 29, NoDCR.
Doze	27	Indicates that the processor was in a low-power mode when a debug exception occurred: Encoding / Meaning 0: Processor not in low-power mode when debug exception occurred 1: Processor in low-power mode when debug exception occurred See the introduction above for corner cases in setting the state of this bit. The Doze bit indicates Reduced Power (RP) and WAIT, and other implementation-dependent low-power modes. If the implementation does not support low-power modes, then this bit always reads as 0.	R	Undefined	Required
Halt	26	Indicates that the internal processor system bus clock was stopped when the debug exception occurred: Encoding / Meaning 0: Internal system bus clock running 1: Internal system bus clock stopped See the introduction above for corner cases in setting the state of this bit. Halt indicates WAIT, and other implementation-dependent events that stop the system bus clock. If the implementation does not support a halt state, then the bit always reads as 0.	R	Undefined	Required

Table 6.13 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
CountDM	25	Controls or indicates the Count register behavior in Debug Mode. Implementations can have fixed behavior, in which case this bit is read-only (R), or the implementation can allow this bit to control the behavior, in which case this bit is read/write (R/W). The reset value of this bit indicates the behavior after reset, and depends on the implementation. Encoding of the bit is: Encoding: 0 — Count register stopped in Debug Mode Encoding: 1 — Count register is running in Debug Mode If not implemented, this bit is read-only (R) and reads as zero.	R or R/W	Preset	Required
IBusEP	24	Indicates if a Bus Error exception is pending from an instruction fetch. Set when an instruction fetch bus error event occurs or a 1 is written to the bit by software. Cleared when a Bus Error exception on an instruction fetch is taken by the processor. If IBusEP is set when IEXI is cleared, a Bus Error exception on an instruction fetch is taken by the processor, and IBusEP is cleared. In Debug Mode, a Bus Error exception applies to a Debug Mode Bus Error exception. If not implemented, this bit is read-only (R) and reads as zero.	R/W1	0	Required if imprecise bus error can occur on instruction fetch, otherwise optional
MCheckP	23	Indicates if a Machine Check exception is pending. Set when a machine check event occurs or a 1 is written to the bit by software. Cleared when a Machine Check exception is taken by the processor. If MCheckP is set when IEXI is cleared, a Machine Check exception is taken by the processor, and MCheckP is cleared. In Debug Mode, a Machine Check exception applies to a Debug Mode Machine Check exception. Note that machine checks due to duplicate TLB entries must be reported asynchronous with respect to the instruction that causes them, and these would be prioritized as "Other execution-based exception" in Table 6.8. In this case this bit would not be set. Any asynchronous implementation-dependent machine check should be reported using EJTAG priority in Table 6.8. If not implemented, this bit is read-only (R) and reads as zero.	R/W1	0	Required if imprecise machine check error can occur, otherwise optional

Encoding table for CountDM:

Encoding	Meaning
0	Count register stopped in Debug Mode
1	Count register is running in Debug Mode

Table 6.13 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
CacheEP	22	Indicates if a Cache Error is pending. Set when a cache error event occurs or a 1 is written to the bit by software. Cleared when a Cache Error exception is taken by the processor. If CacheEP is set when IEXI is cleared, a Cache Error exception is taken by the processor, and CacheEP is cleared. In Debug Mode, a Cache Error exception applies to a Debug Mode Cache Error exception. If not implemented, this bit is read-only (R) and reads as zero.	R/W1	0	Required if imprecise cache error can occur, otherwise optional
DBusEP	21	Indicates if a Data Access Bus Error exception is pending. Set when a data access bus error event occurs or a 1 is written to the bit by software. Cleared when a Bus Error exception on data access is taken by the processor. If DBusEP is set when IEXI is cleared, a Bus Error exception on data access is taken by the processor, and DBusEP is cleared. In Debug Mode, a Bus Error exception applies to a Debug Mode Bus Error exception. If not implemented, this bit is read-only (R) and reads as zero.	R/W1	0	Required if imprecise bus error can occur on data access, otherwise optional
IEXI	20	An Imprecise Error eXception Inhibit (IEXI) controls exceptions taken due to imprecise error indications. Set when the processor takes a debug exception or an exception in Debug Mode occurs. Cleared by execution of the DERET instruction. Otherwise modifiable by Debug Mode software. When IEXI is set, then the imprecise error exceptions from bus errors on instruction fetches or data accesses, cache errors, or machine checks are inhibited and deferred until the bit is cleared. If not implemented, this bit is read-only (R) and reads as zero.	R/W	0	Required if any imprecise error covered by MCheckP, CacheEP, IBusEP or DBusEP, can occur, otherwise optional
DDBSImpr	19	Indicates that a Debug Data Break Store Imprecise exception due to a store was the cause of the debug exception, or that an imprecise data hardware break due to a store was indicated after another debug exception occurred. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0: No match of an imprecise data hardware breakpoint on store 1: Match of imprecise data hardware breakpoint on store If not implemented, this bit reads as zero.	R	Undefined	Required if Debug Data Break on Store Imprecise exception can occur, otherwise optional

Table 6.13 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
DDBLImpr	18	Indicates that a Debug Data Break Load Imprecise exception due to a load was the cause of the debug exception, or that an imprecise data hardware break due to a load was indicated after another debug exception occurred. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0 — No match of an imprecise data hardware breakpoint on load 1 — Match of imprecise data hardware breakpoint on load If not implemented, this bit reads as zero.	R	Undefined	Required if Debug Data Break on Load Imprecise exception can occur, otherwise optional
EJTAGver	17:15	Provides the EJTAG version. Note that each new version number is used to indicate the addition of a significant new modification or addition to the architecture. For example, Version 3.1 (value of 3) indicates the EJTAG upgrade that includes PC sampling. Similarly, Version 4.0 (value 4) includes the addition of Complex Break and Trigger (CBT) feature. A processor or core that implements PC sampling should indicate a version number of 3. Intermediate revisions of the specification only include typographical edits and address minor issues in the specification itself without adding any new features. It is recommended that an implementation use the latest version of the specification since features like PC sampling and CBT are optional. **Encoding** / **Meaning** 0 — Version 1 and 2.0 1 — Version 2.5 2 — Version 2.6 3 — Version 3.1 4 — Version 4.0 5-7 — Reserved	R	Preset	Required
DExcCode	14:10	Indicates the cause of the latest exception in Debug Mode. The field is encoded as the ExcCode field in the Cause register for those exceptions that can occur in Debug Mode (the encoding is shown in MIPS32 and MIPS64 specifications), with addition of code 30 with the mnemonic CacheErr for cache errors and the use of code 9 with mnemonic Bp for the SDBBP instruction. This value is undefined after a debug exception.	R	Undefined	Required

Table 6.13 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
NoSSt	9	Indicates whether the single-step feature controllable by the SSt bit is available in this implementation: **Encoding** / **Meaning** 0: Single-step feature available 1: No single-step feature available A minimum number of hardware instruction breakpoints must be available if no single-step feature is implemented in hardware. Refer to Section 3.8.1 on page 58 for more information.	R	Preset	Required
SSt	8	Controls whether single-step feature is enabled: **Encoding** / **Meaning** 0: No enable of single-step feature 1: Single-step feature enabled If not implemented due to no single-step feature (NoSSt is 1), this bit is read-only (R) and reads as zero. If implemented, then in a processor with MIPS MT, this bit is instantiated on a per-TC basis.	R/W	0	Required if single-step features are available, otherwise not implemented
OffLine	7	In MIPS MT processors, this bit is instantiated on a per-TC basis and allows a hardware thread context (TC) to be taken off-line for debug. **Encoding** / **Meaning** 0: TC may fetch and issue according to the rules of MIPS MT 1: TC may only fetch and execute in Debug mode. In non-MT processors, the OffLine bit, if implemented, inhibits the fetch and issue of instructions by the processor as a whole, unless it is in Debug mode. This allows isolation of processors in a multi-processor or multi-core system. Following a DERET with the OffLine bit set, a MIPS MT processor can be taken out of the off-line state by a MTTR instruction targeting the off-line TC's Debug register, by a DINT Debug exception handler, or a hardware reset. Following a DERET with the OffLine bit set, a non-MT processor can only be taken out of the off-line state by a DINT Debug exception handler clearing the OffLine bit, or a hardware reset. If not implemented, this bit is read-only (R) and reads as zero.	R/W	0	Required for processors implementing EJTAG and MIPS MT ASE. Otherwise optional.

Table 6.13 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
DIBImpr	6	Indicates that a Debug Instruction Break Imprecise exception occurred. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0 — No Debug Instruction Break Imprecise exception 1 — Debug Instruction Break Imprecise exception If not implemented, this bit reads as zero.	R	Undefined	Required if Debug Instruction Break Imprecise exception can occur, otherwise optional
DINT	5	Indicates that a Debug Interrupt exception occurred. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0 — No Debug Interrupt exception 1 — Debug Interrupt exception If not implemented, this bit reads as zero.	R	Undefined	Required if Debug Interrupt exception can occur, otherwise not implemented
DIB	4	Indicates that a Debug Instruction Break exception occurred. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0 — No Debug Instruction Break exception 1 — Debug Instruction Break exception If not implemented, this bit reads as zero.	R	Undefined	Required if Debug Instruction Break exception can occur, otherwise not implemented
DDBS	3	Indicates that a Debug Data Break Store exception occurred on a store due to a precise data hardware break. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0 — No Debug Data Break Store Exception 1 — Debug Data Break Store Exception If not implemented, this bit reads as zero.	R	Undefined	Required if Debug Data Break Store exception can occur, otherwise not implemented
DDBL	2	Indicates that a Debug Data Break Load exception occurred on a load due to a precise data hardware break. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0 — No Debug Data Break Store Exception 1 — Debug Data Break Store Exception If not implemented, this bit reads as zero.	R	Undefined	Required if Debug Data Break Load exception can occur, otherwise not implemented

Table 6.13 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
DBp	1	Indicates that a Debug Breakpoint exception occurred. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0 — No Debug Breakpoint exception 1 — Debug Breakpoint exception	R	Undefined	Required
DSS	0	Indicates that a Debug Single Step exception occurred. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0 — No debug single-step exception 1 — Debug single-step exception This bit is read-only (R) and reads as zero if not implemented. On a processor implementing the MIPS MT, this bit is implemented per-VPE.	R	Undefined	Required if Debug Single Step exception can occur, otherwise not implemented

6.7.2 Debug2 Register (CP0 Register 23, Select 6)

Compliance Level: Required for EJTAG debug support for EJTAG specification 4.00 and higher.

The Debug2 register is a read/write register that is used to indicate the cause of debug exceptions due to complex breakpoints if implemented. The size of this register is 32 bits for 32-bit processors and 64 bits for 64-bit processor.

Figure 6.9 shows the format of the DESAVE register; Table 6.16 describes the DESAVE register field.

Figure 6.7 Debug2 Register Format

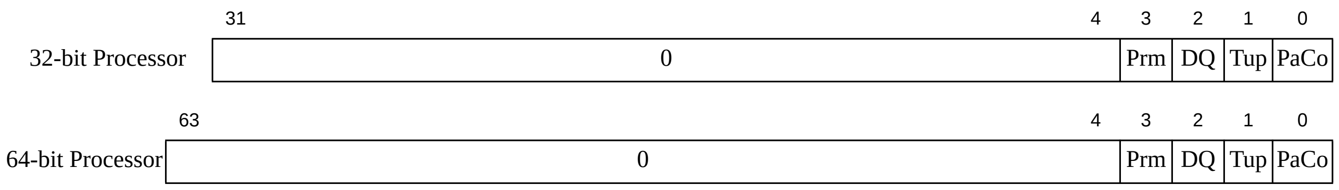


Table 6.14 Debug2 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	**Bits**				
Prm	3	This bit indicates that the break exception happened due to a primed complex break match. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0: No Debug Primed Break exception 1: Debug Primed Break exception If not implemented, this bit reads as zero.	R	Undefined	Required if primed break is supported $CBTC_{PP}$=1
DQ	2	This bit indicates that the break exception happened due to a data qualified complex break match. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0: No Debug Data Qualified Break exception 1: Debug Data Qualified Break exception If not implemented, this bit reads as zero.	R	Undefined	Required if data qualified break is supported $CBTC_{DQP}$=1
Tup	1	This bit indicates that the break exception happened due to a tuple complex break match. Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0: No Debug Tuple Break exception 1: Debug Tuple Break exception If not implemented, this bit reads as zero.	R	Undefined	Required if tuple break is supported $CBTC_{TP}$=1
PaCo	0	This bit indicates that the break exception happened when a pass counter in the complex break unit reached a zero count (this overrides other settings on the breakpoint such as data qualifier or prime condition). Cleared on exception in Debug Mode. **Encoding** / **Meaning** 0: No Debug Instruction, Data, or Tuple Break on pass counter exception 1: Debug Instruction, Data, or Tuple Break on pass counter exception If not implemented, this bit reads as zero.	R	Undefined	Required if pass counter is supported $CBTC_{PCP}$=1
0	MSB:4	Must be written as zeros return zeros on reads.	0	0	Reserved

6.7.3 Debug Exception Program Counter Register (CP0 Register 24, Select 0)

Compliance Level: Required for EJTAG debug support.

The Debug Exception Program Counter (DEPC) register is a read/write register that contains the address at which processing resumes after the exception has been serviced. The size of this register is 32 bits for 32-bit processors and

64 bits for 64-bit processors, even with only 32-bit virtual addressing enabled. All bits of the DEPC register are significant and writable. A DMFC0 from the DEPC register returns the full 64-bit DEPC on 64-bit processors.

Hardware updates this register on debug exceptions and exceptions in Debug Mode.

For precise debug exceptions and precise exceptions in Debug Mode, the DEPC register contains either:

- the virtual address of the instruction that was the direct cause of the exception, or
- the virtual address of the immediately preceding branch or jump instruction, when the exception-causing instruction is in a branch delay slot, and the Debug Branch Delay (BDB) bit in the Debug register is set.

For imprecise debug exceptions and imprecise exceptions in Debug Mode, the DEPC register contains the address at which execution is resumed when returning to Non-Debug Mode.

Figure 6.8 shows the format of the DEPC register; Table 6.15 describes the DEPC register field.

Figure 6.8 DEPC Register Format

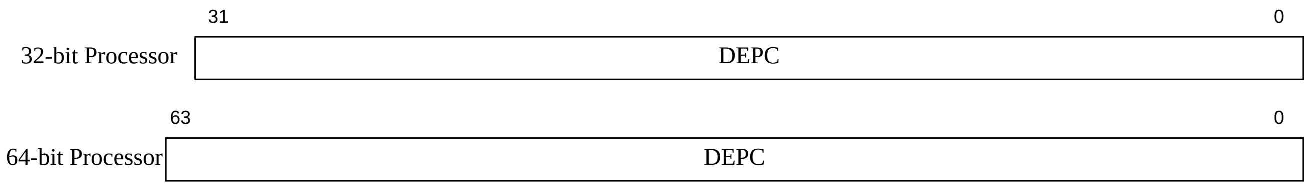


Table 6.15 DEPC Register Field Description

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
DEPC	MSB:0	Debug Exception Program Counter	R/W	Undefined	Required

6.7.4 Debug Exception Save Register (CP0 Register 31, Select 0)

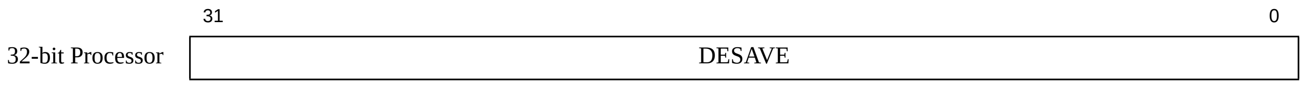
Compliance Level: Required for EJTAG debug support.

The Debug Exception Save (DESAVE) register is a read/write register that functions as a simple scratchpad register. The size of this register is 32 bits for 32-bit processors and 64 bits for 64-bit processor.

The debug exception handler uses this to save one of the GPRs, which is then used to save the rest of the context to a pre-determined memory area, for example, in the dmseg segment. This register allows the safe debugging of exception handlers and other types of code where the existence of a valid stack for context saving cannot be assumed.

Figure 6.9 shows the format of the DESAVE register; Table 6.16 describes the DESAVE register field.

Figure 6.9 DESAVE Register Format



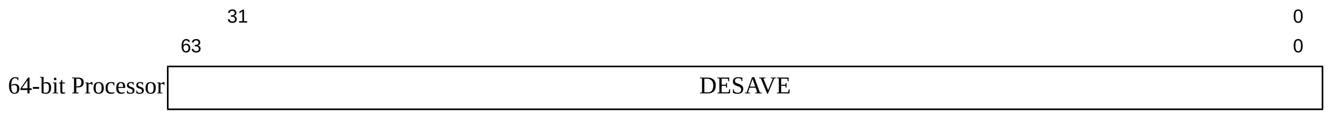


Table 6.16 DESAVE Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
DESAVE	MSB:0	Debug Exception Save contents	R/W	Undefined	Required

6.8 EJTAG Instructions

The SDBBP and DERET instructions are added to the processor's instruction set as part of the required EJTAG features. These instructions are described on the next two pages.

31 26	25 6	5 0
SPECIAL2 011100	code	SDBBP 111111
6	20	6

15 11	10 5	4 0	
RR 11101	code	SDBBP 00001	**MIPS16e** Format
5	6	5	

Format: `SDBBP code` **EJTAG**

Purpose: Software Debug Breakpoint

To cause a debug breakpoint exception

Description:

This instruction causes a debug exception, passing control to the debug exception handler. If the processor is executing in Debug Mode when the SDBBP instruction is executed, the exception is a Debug Mode Exception, which sets the Debug$_{DExcCode}$ field to the value 0x9 (Bp). The code field can be used for passing information to the debug exception handler, and is retrieved by the debug exception handler only by loading the contents of the memory word containing the instruction, using the *DEPC* register. The *CODE* field is not used in any way by the hardware.

Restrictions:

A Reserved Instruction Exception is signaled if EJTAG is not implemented.

Operation:

```
If Debug_DM = 0 then
   SignalDebugBreakpointException()          /* See Section 6.3.3 on page 93 */
else
   SignalDebugModeBreakpointException()      /* See Section 6.4.3 on page 101 */
endif
```

Exceptions:

Debug Breakpoint exception
Debug Mode Breakpoint exception

31 … 26	25	24 … 6	5 … 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	DERET 011111
6	1	19	6

Format: DERET **EJTAG**

Purpose: Debug Exception Return

To Return from a debug exception.

Description:

DERET clears execution and instruction hazards, returns from Debug Mode and resumes non-debug execution at the instruction whose address is contained in the *DEPC* register. DERET does not execute the next instruction (i.e. it has no delay slot).

Restrictions:

A DERET placed between an LL and SC instruction does not cause the SC to fail.

If the DEPC register with the return address for the DERET was modified by an MTC0 or a DMTC0 instruction, a CP0 hazard exists that must be removed via software insertion of the appropriate number of SSNOP instructions (for implementations of Release 1 of the Architecture) or by an EHB, or other execution hazard clearing instruction (for implementations of Release 2 of the Architecture).

DERET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the DERET returns.

This instruction is legal only if the processor is executing in Debug Mode, the operation of the processor is **UNDEFINED** otherwise.

The operation of the processor is **UNDEFINED** if a DERET is executed in the delay slot of a branch or jump instruction.

Operation:

```
if $Debug_{DM}$ = 1 then
    $Debug_{DM} \leftarrow 0$
    $Debug_{IEXI} \leftarrow 0$
    if IsMIPS16Implemented() then
        PC $\leftarrow DEPC_{PCWIDTH-1..1} \parallel 0$
        ISAMode $\leftarrow DEPC_0$
    else
        PC $\leftarrow$ DEPC
    endif
else
    UNDEFINED
endif
ClearHazards()
```

Exceptions:

Coprocessor Unusable Exception
Reserved Instruction Exception

Chapter 7

EJTAG Test Access Port

This chapter describes the EJTAG features provided when the optional EJTAG Test Access Port (TAP) is included in the implementation. The TAP is an optional part of EJTAG, but if implemented then it is required that the DCR is also implemented, and all features in the TAP described below are required, except for those features explicitly mentioned as optional.

This chapter contains the following sections:

7.1 TAP Overview

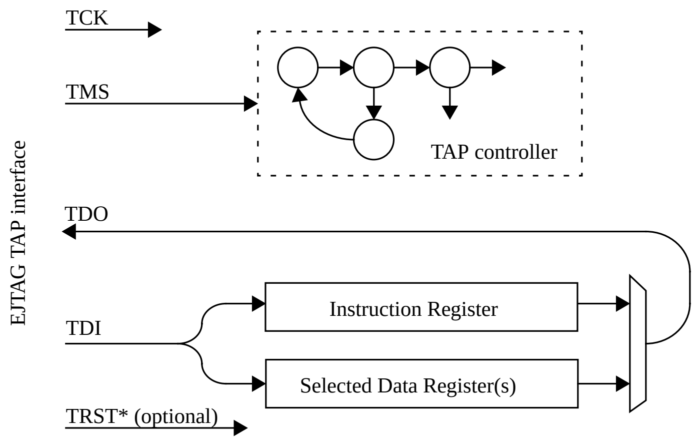
The overall features of the EJTAG Test Access Port (TAP) are:

- Identification of device and EJTAG debug features accessed through the TAP
- dmseg segment memory "emulation" (mapping dmseg segment processor accesses into probe transactions).
- Reset handling allows debug exception immediately after reset
- Debug interrupt request from probe
- Low-power mode indications
- Implementation-dependent processor and peripheral reset

If the TAP is not implemented then other features depending on register values and indications from the TAP should behave as if these register values and indications have the power-up and reset value.

Figure 7.1 shows an overview of the elements in the TAP.

Figure 7.1 Test Access Port (TAP) Overview



The TAP consists of the following signals: Test Clock (TCK), Test Mode (TMS), Test Data In (TDI), Test Data Out (TDO), and the optional Test Reset (TRST*). TCK and TMS control the state of the TAP controller, which controls access to the Instruction or selected data register(s). The Instruction register controls selection of data registers. Access to the Instruction and data register(s) occurs serially through TDI and TDO. The optional TRST* is an asynchronous reset signal to the TAP.

Access through the TAP does not interfere with the operation of the processor, unless features specifically described to do so are used.

The description of the EJTAG TAP in this chapter is intended only to cover EJTAG issues related to use of a TAP. Consult the “IEEE Std 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture” for detailed information about use of a TAP for other purposes, for example, integration with JTAG boundary scan.

For EJTAG features, there are no difference between a reset and a soft reset occurring to the processor; they behave identically in both Debug Mode and Non-Debug Mode. References to reset in the following therefore refers to both reset (hard reset) and soft reset.

7.2 TAP Signals

The signals TCK, TMS, TDI, TDO, and the optional TRST* make up the interface for the TAP. These signals are described in detail below. Refer to Chapter 8, “On-Chip Interfaces” on page 147 for the connection of the signals to chip pins.

7.2.1 Test Clock Input (TCK)

TCK is the clock that controls the updating of the TAP controller and the shifting of data through the Instruction or selected data register(s).

TCK is independent of the processor clock, with respect to both frequency and phase.

7.2.2 Test Mode Select Input (TMS)

TMS is the control signal for the TAP controller. This signal is sampled on the rising edge of TCK.

7.2.3 Test Data Input (TDI)

TDI is the test data input to the Instruction or selected data register(s). This signal is sampled on the rising edge of TCK for some TAP controller states.

7.2.4 Test Data Output (TDO)

TDO is the test data output from the Instruction or data register(s). This signal changes on the falling edge of TCK, or becomes 3-stated asynchronously when TRST* is driven low.

The off-chip TDO is only driven when data is shifted out, otherwise the off-chip TDO is 3-stated.

The 3-state notation indicates that the TDO off-chip signal is undriven.

7.2.5 Test Reset Input (TRST*)

TRST* is the optional test reset input that asynchronously resets the TAP, with the following immediate effects:

- The TAP controller is put into the Test-Logic-Reset state
- The Instruction register is loaded with the IDCODE instruction
- Any EJTAGBOOT indication is cleared
- The TDO output is 3-stated

TRST* does not reset another part of the TAP or processor. Thus this type of reset does not affect the processor, and the processor reset is not allowed to have any effect on the above parts of the TAP.

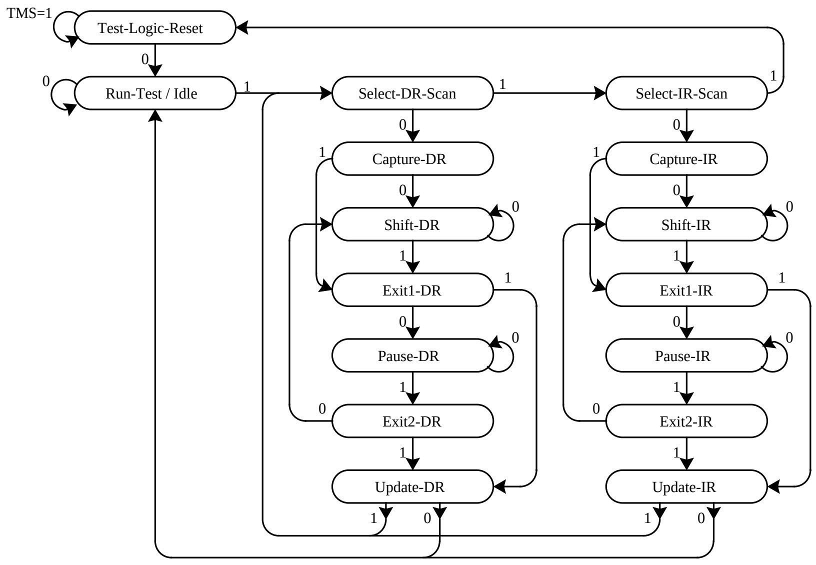
Even though TRST* is an optional signal, the TRST* signal is referred to in the following discussions. If TRST* is not implemented, then a power-up reset of the TAP must provide the reset functionality similar to a low value on TRST* during power-up.

7.3 TAP Controller

The TAP controller is a state machine whose active state controls TAP reset and access to Instruction and data registers.

The state transitions in the TAP controller occur on the rising edge of TCK or when TRST* goes low. The TMS signal determines the transition at the rising edge of TCK. Figure 7.2 shows the state diagram for the TAP controller.

Figure 7.2 TAP Controller State Diagram



The behavior of the functional states shown in the figure is described below. The non-functional states are intermediate states in which no registers in the TAP change, and are not described here.

Events in the following subsections are described with relation to the rising and falling edge of TCK. The described events take place when the TAP controller is in the corresponding state when the clock changes.

The TAP controller is forced into the Test-Logic-Reset state at power-up either by a low value on TRST* or by a power-up reset circuit.

7.3.1 Test-Logic-Reset State

When the Test-Logic-Reset state is entered, the Instruction register is loaded with the IDCODE instruction, and any EJTAGBOOT indication is cleared. This state ensures that the TAP does not interfere with the normal operation of the CPU core.

The TAP controller always reaches this state after five rising edges on TCK when TMS is set to 1.

A low value on TRST* immediately places the TAP controller in this state asynchronous to TCK.

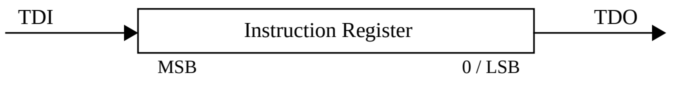
7.3.2 Capture-IR State

In the Capture-IR state, the two LSBs of the Instruction register are loaded with the value 01_2, and the upper MSBs are loaded with implementation-dependent values. Both values are loaded on the rising edge of TCK.

7.3.3 Shift-IR State

In the Shift-IR state, the LSB of the Instruction register is output on TDO on the falling edge of TCK. The Instruction register is shifted one position from MSB to LSB on the rising edge of TCK, with the MSB shifted in from TDI. The value in the Instruction register does not take effect until the Update-IR state. Figure 7.3 shows the shifting direction for the Instruction register.

Figure 7.3 TDI to TDO Path when in Shift-IR State



The length of the Instruction register is specified in Section 7.4 on page 123.

The value loaded in the Capture-IR state is used as the initial value for the Instruction register when shifting starts; thus it is not possible to read out the previous value of the Instruction register.

7.3.4 Update-IR State

In the Update-IR state, the value in the Instruction register takes effect on the rising or falling edge of TCK.

7.3.5 Capture-DR State

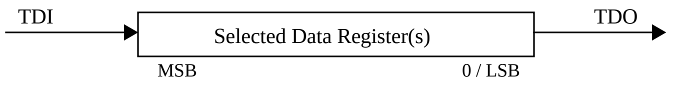
In the Capture-DR state, the value of the selected data register(s) is captured on the rising edge of TCK for shifting out in the Shift-DR state. The Capture-DR state reads the data, in order to output this read value in the Shift-DR state.

The Instruction register controls the selection of the following data register(s): Bypass, Device ID, Implementation, EJTAG Control, Address, and Data register(s).

7.3.6 Shift-DR State

In the Shift-DR state, the LSB of the selected data register(s) is output on TDO on the falling edge of TCK. The selected data register(s) is shifted one position from MSB to LSB on the rising edge of TCK, with TDI shifted in at the MSB. The value(s) shifted into the register(s) does not take effect until the Update-DR state. Figure 7.4 shows the shifting direction for the selected data register.

Figure 7.4 TDI to TDO Path for Selected Data Register(s) when in Shift-DR State



The length of the shift path depends on the selected data register(s).

7.3.7 Update-DR State

In the Update-DR state, the update of the selected data register(s) with the value from the Shift-DR state occurs on the falling or rising edge of TCK. This update writes the selected register(s).

7.4 Instruction Register and Special Instructions

The Instruction register controls selection of accessed data register(s), and controls the setting and clearing of the EJTAGBOOT indication.

The Instruction register is five or more bits wide when used with EJTAG. Table 7.1 shows the allocation of the TAP instruction.

Table 7.1 TAP Instruction Overview

Code	Instruction	Function
All 0's	(Free for other use)	Free for other use, such as JTAG boundary scan
0x01	IDCODE	Selects Device Identification (ID) register
0x02	(Free for other use)	Free for other use, such as JTAG boundary scan
0x03	IMPCODE	Selects Implementation register
0x04 - 0x07	(Free for other use)	Free for other use, such as JTAG boundary scan
0x08	ADDRESS	Selects Address register
0x09	DATA	Selects Data register
0x0A	CONTROL	Selects EJTAG Control register
0x0B	ALL	Selects the Address, Data and EJTAG Control registers
0x0C	EJTAGBOOT	Makes the processor take a debug exception after reset
0x0D	NORMALBOOT	Makes the processor execute the reset handler after reset
0x0E	FASTDATA	Selects the Data and Fastdata registers
0x0F	(EJTAG reserved)	Reserved for future EJTAG use
0x10	TCBCONTROLA	Selects the control register *TCBTraceControl* in the Trace Control Block
0x11	TCBCONTROLB	Selects another trace control block register
0x12	TCBDATA	Used to access the registers specified by the $\text{TCBCONTROLB}_{\text{REG}}$ field and transfers data between the TAP and the TCB control register
0x13	TCBCONTROLC	Selects another trace control block register
0x14	PCSAMPLE	Selects the PCsample register
0x15	TCBCONTROLD	Selects another trace control block register
0x16	TCBCONTROLE	Selects another trace control block register
0x17 - 0x1B	(EJTAG reserved)	Reserved for future EJTAG use
0x1C - All 1's	(Free for other use)	Free for other use, such as JTAG boundary scan
All 1's	BYPASS	Select Bypass register

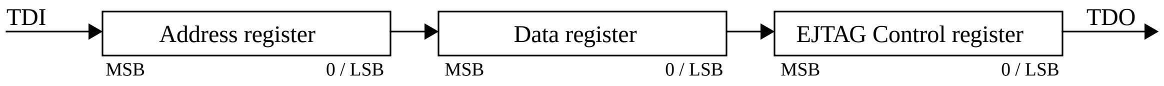
The instructions IDCODE, IMPCODE, ADDRESS, DATA, CONTROL, and BYPASS select a single data register, as indicated in the table. The unused instructions reserved for EJTAG select the Bypass register. The ALL, EJTAGBOOT, NORMALBOOT, and FASTDATA instructions are described in the following subsections. The instructions that are related to trace registers in the trace control block (TCB) are described in the Trace Control Block Specification document.

Any EJTAGBOOT indication is cleared at power-up either by a low value on the TRST* or by a power-up reset circuit, and the Instruction register is loaded with the IDCODE instruction.

7.4.1 ALL Instruction

The Address, Data and EJTAG Control data registers are selected at once with the ALL instruction, as shown in Figure 7.5.

Figure 7.5 TDI to TDO Path when in Shift-DR State and ALL Instruction is Selected



7.4.2 EJTAGBOOT and NORMALBOOT Instructions

The EJTAGBOOT and NORMALBOOT instructions control whether a debug interrupt is requested as a result of a reset. If EJTAGBOOT is indicated then a debug interrupt is requested at reset, and a Debug Interrupt exception is taken after the processor is reset, and instead of fetching instructions from the reset exception vector, instructions are fetched from the debug exception vector. The location of the debug exception vector is controlled by the ProbTrap bit in the EJTAG Control register (see Table 7.9 on page 134). If the ProbTrap bit is set, the debug exception handler is in this case fetched from the probe through the dmseg segment. It is possible to take the debug exception and execute the debug handler from the probe even if no instructions can be fetched from the reset handler. This condition guarantees that the system will not hang at reset when the EJTAGBOOT feature is used, even if the normal memory system does not work properly.

An internal EJTAGBOOT indication holds information on the action to take at a processor reset, and this is set when the EJTAGBOOT instruction takes effect in the Update-IR state. The indication is cleared when the NORMALBOOT instruction takes effect in the Update-IR state, or when the Test-Logic-Reset state is entered, for example, when TRST* is asserted low. The requirement of clearing the internal EJTAGBOOT indication when the Test-Logic-Reset state is entered, and not on a TCK clock when in the state, ensures that the indication can be cleared with five clocks on TCK when TMS is high.

The internal EJTAGBOOT indication is cleared at power-up either by a low value on the TRST* or by a power-up reset circuit. Thus the processor executes the reset handler after power-up unless the EJTAGBOOT instruction is given through the TAP.

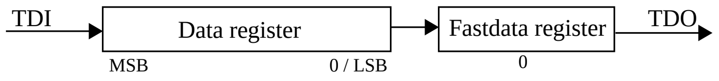
The Bypass register is selected when the EJTAGBOOT or NORMALBOOT instruction is given.

The EjtagBrk, ProbEn, and ProbTrap bits in the EJTAG Control register follow the internal EJTAGBOOT indication. They are all set at processor reset if a Debug Interrupt exception is to be generated, with execution of the debug handler from the probe.

7.4.3 FASTDATA Instruction

This selects the Data and the Fastdata registers at once, as shown in Figure 7.6. The use of the FASTDATA instruction is described in more detail in Section 7.5.6 “Fastdata Register (TAP Instruction FASTDATA)”.

Figure 7.6 TDI to TDO Path when in Shift-DR State and FASTDATA Instruction is Selected



7.5 TAP Data Registers

Table 7.2 summarizes the data registers in the TAP. Complete descriptions of these registers are located in the following subsections.

Table 7.2 EJTAG TAP Data Registers

Instruction Used to Access Register	Register Name	Function	Reference	Compliance Level
IDCODE	Device ID	Identifies device and accessed processor in the device.	See Section 7.5.1 on page 127	Required
IMPCODE	Implementation	Identifies main debug features implemented and accessible through the TAP.	See Section 7.5.2 on page 128	Required
DATA, ALL, or FASTDATA	Data	Data register for processor access.	See Section 7.5.3 on page 130	Required
ADDRESS or ALL	Address	Address register for processor access.	See Section 7.5.4 on page 132	Required
CONTROL or ALL	EJTAG Control	Control register for most EJTAG features used through the TAP.	See Section 7.5.5 on page 133	Required
BYPASS, EJTAGBOOT, NORMALBOOT, or unused EJTAG instructions	Bypass	Provides a one bit shift path through the TAP.	See Section 7.5.8 on page 141	Required
FASTDATA	Fastdata	Provides a one bit register whose value is tagged to the front of the Data register to capture the value of the processor access pending (PrAcc) bit in the EJTAG Control register	See Section 7.4.3 on page 125	Required with EJTAG version 02.60 and higher
TCBCONTROLA	TCBControlA	Implemented and used in the Trace Control Block (TCB). Used by external probe (debugger) software to control tracing output from the core	See the TCB documentation	Required with EJTAG version 02.60 and higher if trace logic is implemented
TCBCONTROLB	TCBControlB	Implemented and used in the Trace Control Block (TCB). Controls tracing configuration options	See the TCB documentation	Required with EJTAG version 02.60 and higher if trace logic is implemented
TCBDATA	TCBData	Implemented and used in the TCB.	See the TCB documentation	Required with EJTAG version 02.60 and higher if trace logic is implemented

Table 7.2 EJTAG TAP Data Registers (Continued)

Instruction Used to Access Register	Register Name	Function	Reference	Compliance Level
TCBCONTROLC	TCBControlC	Implemented and used in the Trace Control Block (TCB). Controls tracing configuration options	See the TCB documentation	Required with EJTAG version 3.10 and higher if trace logic is implemented
PCSAMPLE	PCsample	Implemented and used by the PC Sampling logic	See Chapter 5, "PC Sampling" on page 79.	Optional feature (defined EJTAG 3.10)
TCBCONTROLD	TCBControlD	Implemented and used in the Trace Control Block (TCB). Controls tracing configuration options	See the TCB documentation	Required with EJTAG version 4.10 and higher if trace logic is implemented
TCBCONTROLE	TCBControlE	Implemented and used in the Trace Control Block (TCB). Controls tracing configuration options	See the TCB documentation	Required with EJTAG version 4.10 and higher if trace logic is implemented

A read of a data register corresponds only to the Capture-DR state of the TAP controller, and a write of the data register corresponds to the Update-DR state only.

The initial states of these registers are specified with either a reset state or a power-up state. If a reset state is specified, then the indicated value is applied to the register when a processor reset is applied. If a power-up state is specified, then the indicated value is applied at power-up reset.

TCK does not have to be running in order for a processor reset to reset the registers.

7.5.1 Device Identification (ID) Register (TAP Instruction IDCODE)

Compliance Level: Required with EJTAG TAP feature.

The Device ID register is a 32-bit read-only register that identifies the specific device implementing EJTAG. This register is also defined in IEEE 1149.1. The Device ID register holds a unique number among different devices with EJTAG compliant processors implemented. It is recommended that the register is also unique amongst different EJTAG compliant processors in the same device.

Figure 7.7 shows the format of the Device ID register; Table 7.3 describes the Device ID register fields.

Figure 7.7 Device ID Register Format

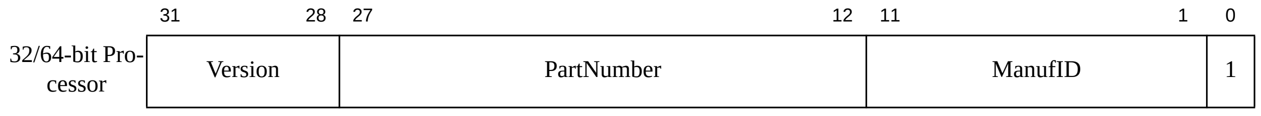


Table 7.3 Device ID Register Field Descriptions

Fields		Description	Read / Write	Power-up State	Compliance
Name	Bits				
Version	31:28	Identifies the version of a specific device. The value in this field must be unique for particular values of Manufacturer ID and Part Number values. The value identifies a specific revision of the design (such as a sequence of bug fixes within the same major design). The value is assigned by the design house.	R	Preset	Required
Part-Number	27:12	Identifies the part number of a specific device. The value in this field must be unique for a particular Manufacturer ID value. Design houses which wish to use the MIPS Technologies, Inc. Manufacturer ID may request assignment of a group of Part Numbers which are then managed by that design house. Assignment of Part Numbers within another Manufacturer ID value is done by the owner of that Manufacturer ID.	R	Preset	Required
ManufID	11:1	Identifies the manufacturer identity code of a specific device, which identifies the design house implementing the processor. According to IEEE 1149.1-1990 section 11.2, the manufacturer identity code is a compressed form of a JEDEC standard manufacturer’s identification code in the JEDEC Publications 106, which can be found at: http://www.jedec.org/ ManufID[6:0] are derived from the last byte of the JEDEC code with the parity bit discarded. ManufID[10:7] provide a binary count of the number of bytes in the JEDEC code that contain the continuation character (0x7F). When the number of continuations characters exceeds 15, these four bits contain the modulo-16 count of the number of continuation characters. If the design house does not have a JEDEC Standard Manufacturer's Identification Code, which is encoded for use in this field, the design house can request use of the MIPS Technologies, Inc. assigned number, or use the number assigned to the core provider. Use of the MIPS Technologies, Inc. number requires prior approval of the Director, MIPS Architecture. The MIPS Technologies, Inc. Standard Manufacturer's Identification Code is 0x127.	R	Preset	Required
1	0	Ignored on write; returns one on read.	R	1	Required

7.5.2 Implementation Register (TAP Instruction IMPCODE)

Compliance Level: Required with EJTAG TAP feature.

The Implementation register is a 32-bit read-only register that identifies features implemented in this EJTAG compliant processor, mainly those accessible from the TAP.

Figure 7.8 shows the format of the Implementation register; Table 7.4 describes the Implementation register fields.

Figure 7.8 Implementation Register Format

	31 29	28	27 25	24	23	22 21	20 17	16	15	14	13 1	0
32/64-bit Processor	EJTAGver	R4k/ R3k	0	DIN T sup	0	ASID size	0	MIPS 16	0	No DMA	0	MIPS 32/64

Table 7.4 Implementation Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Power-up State**	**Compliance**
EJTAGver	31:29	Indicates the EJTAG version: **Encoding** | **Meaning** 0 | Version 1 and 2.0 1 | Version 2.5 2 | Version 2.6 3 | Version 3.1 4 | Version 4.0 5-7 | Reserved	R	Preset	Required
R4k/R3k	28	Indicates R4k or R3k privileged environment: **Encoding** | **Meaning** 0 | R4k privileged environment 1 | R3k privileged environment	R	Preset	Required
DINTsup	24	Indicates support for DINT signal from probe: **Encoding** | **Meaning** 0 | DINT signal from the probe is not supported by this processor 1 | Probe can use DINT signal to make debug interrupt on this processor	R	Preset	Required
ASIDsize	22:21	Indicates size of the ASID field: **Encoding** | **Meaning** 0 | No ASID in implementation 1 | 6-bit ASID 2 | 8-bit ASID 3 | Reserved	R	Preset	Required
MIPS16e	16	Indicates MIPS16e™ ASE support in the processor: **Encoding** | **Meaning** 0 | No MIPS16e support 1 | MIPS16e is supported	R	Preset	Required

Table 7.4 Implementation Register Field Descriptions (Continued)

Fields		Description	Read / Write	Power-up State	Compliance
Name	Bits				
NoDMA	14	Indicates no EJTAG DMA support: **Encoding** | **Meaning** 0 | Reserved 1 | No EJTAG DMA support	R	1	Required
MIPS32/64	0	Indicates 32-bit or 64-bit processor: **Encoding** | **Meaning** 0 | 32-bit processor 1 | 64-bit processor See the R4k/R3k bit for indication of privileged environment.	R	Preset	Required
0	27:25, 23, 20:17, 15, 13:1	Ignored on writes; return zeros on reads.	R	0	Required

7.5.3 Data Register (TAP Instruction DATA, ALL, or FASTDATA)

Compliance Level: Required with EJTAG TAP feature.

The read/write Data register is used for opcode and data transfers during processor accesses. The width of the Data register is 32 bits for 32-bit processors and 64 bits for 64-bit processor.

The value read in the Data register is valid only if a processor access for a write is pending, in which case the data register holds the store value. The value written to the Data register is only used if a processor access for a pending read is finished afterwards, in which case the data value written is the value for the fetch or load. This behavior implies that the Data register is not a memory location where a previously written value can be read afterwards.

Figure 7.9 shows the format of the Data register; Table 7.5 describes the Data register field.

Figure 7.9 Data Register Format

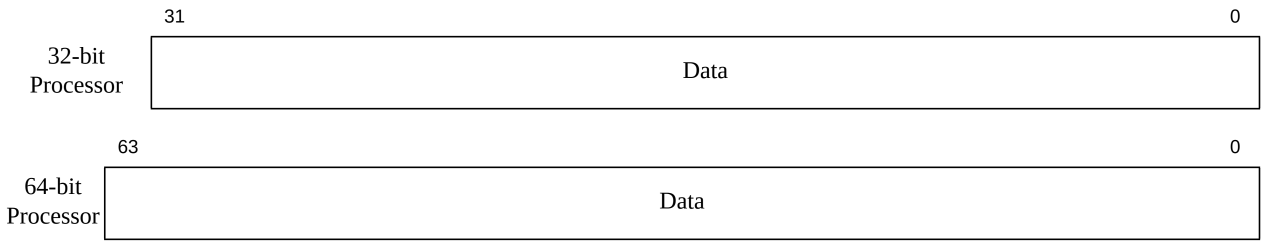


Table 7.5 Data Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
Data	MSB:0	Data used by processor access.	R/W	Undefined	Required

The contents of the Data register are not aligned but hold data as it is seen on a data bus for an external memory system. Thus the bytes are positioned in the Data register based on access size, address, and endianess.

The bytes not accessed for a processor access write are undefined, and the bytes not accessed for a processor access read can be written with any value by the probe shifting the value into the Data register.

Table 7.6 and Table 7.7 show the position of bytes in the Data register for all possible accesses. This positioning depends on the Psz field from the EJTAG Control register, the two or three LSBs from the Address register, and the endianess.

The endianness for Debug Mode, used in the following, is indicated through the ENM bit in the Debug Control Register (DCR), see Chapter 2, “Debug Control Register” on page 29.

Table 7.6 shows the byte positioning for a 32-bit processor (MIPS32/64 = 0), in which case the two LSBs of the Address register are used. Byte 0 refers to bits 7:0, byte 1 refers to bits 15:8, byte 2 refers to bits 23:16, and byte 3 refers to bits 31:24, independent of endianess.

Table 7.6 Data Register Contents for 32-bit Processors

Psz from ECR	Size	Address[1:0]	Little Endian				Big Endian			
			3	2	1	0	3	2	1	0
0	Byte	00_2				▨	▨			
		01_2			▨			▨		
		10_2		▨					▨	
		11_2	▨							▨
1	Halfword	00_2			▨	▨	▨	▨		
		10_2	▨	▨					▨	▨
2	Word	00_2	▨	▨	▨	▨	▨	▨	▨	▨
3	Triple	00_2		▨	▨	▨	▨	▨	▨	
		01_2	▨	▨	▨			▨	▨	▨
Reserved			n.a.				n.a.			

Table 7.7 shows the byte positioning for a 64-bit processor (MIPS32/64 = 1), in which case the three LSBs of the Address register are used. Byte 0 refers to bits 7:0, byte 1 refers to bits 15:8, and so on up to byte 7 which refers to bits 63:56, independent of endianess.

Table 7.7 Data Register Contents for 64-bit Processors

Psz from ECR	Size	Address[2:0]	Little Endian								Big Endian							
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	Byte	000_2								▨	▨							
		001_2							▨			▨						
		010_2						▨					▨					
		011_2					▨							▨				
		100_2				▨									▨			
		101_2			▨											▨		
		110_2		▨													▨	
		111_2	▨															▨
1	Halfword	000_2							▨	▨	▨	▨						
		010_2					▨	▨					▨	▨				
		100_2			▨	▨									▨	▨		
		110_2	▨	▨													▨	▨
2	Word	000_2					▨	▨	▨	▨	▨	▨	▨	▨				
	5-byte/Quinti	001_2				▨	▨	▨	▨	▨	▨	▨	▨	▨	▨			
	6-byte/Sexti	010_2			▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨		
	7-byte/Septi	011_2		▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	
	Word	100_2	▨	▨	▨	▨									▨	▨	▨	▨
	5-byte/Quinti	101_2	▨	▨	▨	▨	▨							▨	▨	▨	▨	▨
	6-byte/Sexti	110_2	▨	▨	▨	▨	▨	▨					▨	▨	▨	▨	▨	▨
	7-byte/Septi	111_2	▨	▨	▨	▨	▨	▨	▨			▨	▨	▨	▨	▨	▨	▨
3	Triple	000_2						▨	▨	▨	▨	▨	▨					
		010_2					▨	▨	▨			▨	▨	▨				
		100_2		▨	▨	▨									▨	▨	▨	
		110_2	▨	▨	▨											▨	▨	▨
	Doubleword	111_2	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
Reserved			n.a.								n.a.							

7.5.4 Address Register (TAP Instruction ADDRESS or ALL)

Compliance Level: Required with EJTAG TAP feature.

The read-only Address register provides the address for a processor access. The width of the register corresponds to the size of the physical address in the processor implementation (from 32 to 64 bits). The specific length is determined by shifting through the Address register, because the length is not indicated elsewhere.

The value read in the register is valid if a processor access is pending, otherwise the value is undefined.

The two or three LSBs of the register are used with the Psz field from the EJTAG Control register to indicate the size and data position of the pending processor access transfer. These bits are not taken directly from the address referenced by the load/store. See Section 7.5.3 on page 130 for more details.

Figure 7.10 shows the format of the Address register; Table 7.8 describes the Address register field.

Figure 7.10 Address Register Format

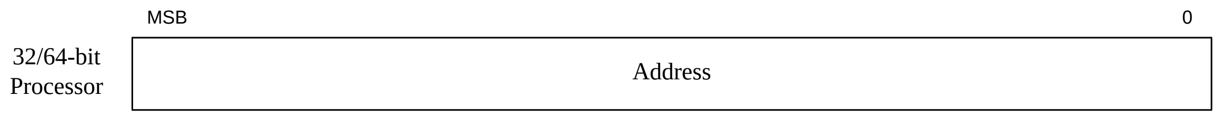


Table 7.8 Address Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Address	MSB:0	Address used by processor access.	R	Undefined	Required

7.5.5 EJTAG Control Register (ECR) (TAP Instruction CONTROL or ALL)

Compliance Level: Required with EJTAG TAP feature.

The 32-bit EJTAG Control Register (ECR) handles processor reset and soft reset indication, Debug Mode indication, access start, finish, and size and read/write indication. The ECR also:

- controls debug vector location and indication of serviced processor accesses,
- allows a debug interrupt request,
- indicates processor low-power mode, and
- allows implementation-dependent processor and peripheral resets.

The EJTAG Control register is not updated/written in the Update-DR state unless the Reset occurred; that is Rocc (bit 31) is either already 0 or is written to 0 at the same time. This condition ensures proper handling of processor accesses after a reset.

Reset of the processor can be indicated through the Rocc bit in the TCK domain a number of TCK cycles after it is removed in the processor clock domain in order to allow for proper synchronization between the two clock domains.

Bits that are R/W in the register return their written value on a subsequent read, unless other behavior is defined. Internal synchronization ensures that a written value is updated for reading immediately afterwards, even when the TAP controller takes the shortest path from the Update-DR to Capture-DR state.

Figure 7.11 shows the format of the EJTAG Control register; Table 7.9 describes the EJTAG Control register fields.

Figure 7.11 EJTAG Control Register Format

	31	30 29	28 24	23	22	21	20	19	18	17	16	15	14	13	12	11 4	3	2 0
32/64-bit Processor	Rocc	Psz	0	VPED	Doze	Halt	PerRst	PRnW	PrAcc	RDVec	PrRst	ProbEn	ProbTrap	0	EjtagBrk	0	DM	0

Table 7.9 EJTAG Control Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	**Bits**				
Rocc	31	Indicates if a processor reset or soft reset has occurred since the bit was cleared: **Encoding** | **Meaning** 0 | No reset occurred 1 | Reset occurred The Rocc bit stays set as long as reset is applied. This bit must be cleared to acknowledge that the reset was detected. The EJTAG Control register is not updated in the Update-DR state unless Rocc is 0 or written to 0 at the same time. This is in order to ensure correct handling of the processor access after reset. Refer to Section 7.6.3 on page 143 for more information on Rocc.	R/W0	1	Required
Psz	30:29	Indicates the size of a pending processor access, in combination with the Address register: **Encoding** | **32-bit Processor MIPS32/64=0** | **64-bit Processor MIPS32/64=1** 0 | Byte | Byte 1 | Halfword | Halfword 2 | Word | Word, 5-7 bytes 3 | Triple | Triple, Doubleword A full description is located in Section 7.5.3 on page 130, including reserved combinations with Address register bits. This field is valid only when a processor access is pending, otherwise the read value is undefined.	R	Undefined	Required
VPED	23	For processors with MIPS MT ASE this bit is a status bit that indicates whether the VPE is currently disabled. A value of 1 indicates that the VPE is disabled and the rest of the EJTAG state is not valid. If this bit is 0, then the processor is either not a MT core or it is an MT core that is currently enabled. Hence, a non-MT core must implement this bit and tie it to zero.	R	0 for non-MT cores and 1 for MT cores	Required for EJTAG version 3.10 and higher.

Table 7.9 EJTAG Control Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
Doze	22	Indicates if the processor is in low-power mode: Encoding | Meaning 0 | Processor is not in low-power mode 1 | Processor is in low-power mode Doze indicates Reduced Power (RP) and WAIT, and other implementation-dependent low-power modes. If the implementation does not support low-power modes, then this bit always reads as 0.	R	0	Required
Halt	21	Indicates if the internal system bus clock is running: Encoding | Meaning 0 | Internal system bus clock is running 1 | Internal system bus clock is stopped Halt indicates WAIT, and other implementation-dependent events that stop the system bus clock. If the implementation does not support a halt state, then the bit always reads as 0.	R	0	Required
PerRst	20	Controls the peripheral reset with implementation-dependent behavior: Encoding | Meaning 0 | No peripheral reset applied 1 | Peripheral reset applied This bit PerRst might not have any effect. There is no inherent indication of whether the PerRst is effective, so the user must consult system documentation. When this bit is changed, then it is only guaranteed that the new value has taken effect when it can be read back here. This handshake mechanism ensures that the setting from the TCK clock domain takes effect in the processor clock domain and in peripherals. This bit is read-only (R) and reads as zero if not implemented.	R/W	0	Optional
PRnW	19	Indicates read or write of a pending processor access: Encoding | Meaning 0 | Read processor access, for a fetch/load access 1 | Write processor access, for a store access This value is defined only when a processor access is pending.	R	Undefined	Required

Table 7.9 EJTAG Control Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
PrAcc	18	Indicates a pending processor access and controls finishing of a pending processor access. When read: Encoding | Meaning 0 | No pending processor access 1 | Pending processor access A write of 0 finishes a processor access if pending; otherwise operation of the processor is UNDEFINED if the bit is written to 0 when no processor access is pending. A write of 1 is ignored. A successful FASTDATA access will clear this bit. See Table 7.11 for details.	R/W0	0	Required
RDVec	17	Controls whether the debug exception vector is relocatable or not. [MORE TEXT HERE]	R/W	0	Required
PrRst	16	Controls the processor reset with implementation-dependent behavior: Encoding | Meaning 0 | No processor reset applied 1 | Processor reset applied The PrRst bit might not have any effect. There is no inherent indication of an effective PrRst, so the user must consult system documentation. If a reset occurs on PrRst, then all parts of the system are reset. It is not allowed for only some device to be reset. When this bit is changed then it is guaranteed that the new value has taken effect when it can be read back here. This handshake mechanism ensures that the setting from the TCK clock domain takes effect in the processor clock domain and in peripherals. However, because a processor reset clears this bit, then the effect of setting it can be that the bit is cleared when the reset takes effect. In this case, the Rocc bit should be observed to detect that the reset took effect. This bit is read-only (R) and reads as zero if not implemented.	R/W	0	Optional

Table 7.9 EJTAG Control Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
ProbEn	15	Controls whether the probe handles accesses to the dmseg segment through servicing of processors accesses: **Encoding** / **Meaning** 0 — Probe will not serve processor accesses 1 — Probe will service processor accesses The ProbEn bit is reflected as a read-only bit in the Debug Control Register (DCR) bit 0, see Chapter 2, “Debug Control Register” on page 29. When this bit is changed, then it is guaranteed that the new value has taken effect in the DCR when it can be read back here. This handshake mechanism ensures that the setting from the TCK clock domain takes effect in the processor clock domain. However, a change of the ProbEn prior to setting the EjtagBrk bit will be effective for the debug handler. Not all combinations of ProbEn and ProbTrap are allowed, see section 7.5.5.2 .	R/W	See Section 7.5.5.1 on page 138	Required
ProbTrap	14	Controls location of the debug exception vector: **Encoding** / **Meaning** 0 — Normal memory 0xFFFF FFFF BFC0 0480 1 — in dmseg at 0xFFFF FFFF FF20 0200 When this bit is changed, then it is guaranteed that the new value is indicated to the processor when it can be read back here. This handshake mechanism ensures that the setting from the TCK clock domain takes effect in the processor clock domain. However, a change of the ProbTrap prior to setting the EjtagBrk bit will be effective at the debug exception. Not all combinations of ProbEn and ProbTrap are allowed, see Section 7.5.5.2 on page 139.	R/W	See Section 7.5.5.1 on page 138	Required

Table 7.9 EJTAG Control Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
EjtagBrk	12	Requests a Debug Interrupt exception to the processor when this bit is written as 1. The debug exception request is ignored if the processor is already in debug at the time of the request. A write of 0 is ignored. The debug request restarts the processor clock if the processor was in a low-power mode. The read value indicates a pending Debug Interrupt exception requested through this bit: **Encoding** / **Meaning**: 0 — No pending Debug Interrupt exception requested through this bit 1 — Pending Debug Interrupt exception The read value can, but is not required to, indicate other pending DINT debug requests (for example, through the DINT signal). This bit is cleared by hardware when the processor enters Debug Mode.	R/W1	See Section 7.5.5.1 on page 138	Required
DM	3	Indicates if the processor is in Debug Mode: **Encoding** / **Meaning**: 0 — Processor is not in Debug Mode 1 — Processor is in Debug Mode	R	0	Required
0	28:24, 17, 13, 11:4, 2:0	Must be written as zeros; return zeros on reads.	0	0	Reserved

7.5.5.1 EJTAGBOOT Indication Determines Reset Value of EjtagBrk, ProbTrap and ProbEn

The reset value of the EjtagBrk, ProbTrap, and ProbEn bits follows the setting of the internal EJTAGBOOT indication. If the EJTAGBOOT instruction has been given, and the internal EJTAGBOOT indication is active, then the reset value of the three bits is set (1), otherwise the reset value is clear (0).

The results of setting these bits are:

- Setting the EjtagBrk causes a Debug Interrupt exception to be requested right after the processor reset from the EJTAGBOOT instruction
- The debug handler is executed from the EJTAG memory because ProbTrap is set to indicate debug vector in EJTAG memory at 0xFFFF FFFF FF20 0200
- Service of the processor access is indicated because ProbEn is set

Thus it is possible to execute the debug handler right after a processor reset from the EJTAGBOOT instruction, without executing any instructions from the normal reset handler.

7.5.5.2 Combinations of ProbTrap and ProbEn

Use of ProbTrap and ProbEn allows independent specification of the debug exception vector location and availability of EJTAG memory. Behavior for the different combinations is shown in Table 7.10. Note that not all combinations are allowed. The second combination shown in the table, that is ProbTrap is 0 and ProbEn is 1, puts the debug handler in memory, but also makes the probe available. This combination can be useful since running from memory could be much faster in general, but the probe can still be accessed to say write the current state of the processor.

Table 7.10 Combinations of ProbTrap and ProbEn

ProbTrap	ProbEn	Debug Exception Vector	Processor Accesses
0	0	Normal memory at 0xFFFF FFFF BFC0 0480	Not serviced by probe
0	1		Serviced by probe
1	0	If these two bits are changed to this state, the operation of the processor is UNDEFINED, indicating that the debug exception vector is in EJTAG memory, but the probe will not service processor accesses.	
1	1	EJTAG memory at 0xFFFF FFFF FF20 0200	Serviced by probe

7.5.6 Fastdata Register (TAP Instruction FASTDATA)

Compliance Level: Required with EJTAG TAP feature for EJTAG version 02.60 and higher.

The width of the Fastdata register is 1 bit. During a Fastdata access, the Fastdata register is written and read, i.e., a bit is shifted in and a bit is shifted out. (See Section 7.4.3 on page 125 for how the Data + Fastdata registers are selected by the FASTDATA instruction.) During a Fastdata access, the Fastdata register value shifted in specifies whether the Fastdata access should be completed or not. The value shifted out is a flag that indicates whether the Fastdata access was successful or not (if completion was requested).

Figure 7.12 Fastdata Register Format

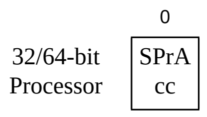


Table 7.11 Fastdata Register Field Description

Fields		Description	Read / Write	Power-up State	Compliance
Name	Bits				
SPrAcc	0	Shifting in a zero value requests completion of the Fastdata access. The PrAcc bit in the EJTAG Control register is overwritten with zero when the access succeeds. (The access succeeds if PrAcc is one and the operation address is in the legal dmseg segment Fastdata area.) When successful, a one is shifted out. Shifting out a zero indicates a Fastdata access failure. Shifting in a one does not complete the Fastdata access and the PrAcc bit is unchanged. Shifting out a one indicates that the access would have been successful if allowed to complete and a zero indicates the access would not have successfully completed.	R/W	Undefined	Required

The FASTDATA access is used for efficient block transfers between the dmseg segment (on the probe) and target memory (on the processor). An “upload” is defined as a sequence of processor loads from target memory and stores to the dmseg segment. A “download” is a sequence of processor loads from the dmseg segment and stores to target memory. The “Fastdata area” is a special range of dmseg segment addresses (0xF..F20.0000 - 0xF..F20.000F) that must be used for Fastdata uploads and downloads. The Data + Fastdata registers (selected with the FASTDATA instruction) allow efficient completion of pending Fastdata area accesses.

During Fastdata uploads and downloads, the processor will stall on accesses to the Fastdata area. The PrAcc (processor access pending bit) will be 1 indicating the probe is required to complete the access. Both upload and download accesses by the probe are attempted by shifting in a zero SPrAcc value (to request access completion) and shifting out SPrAcc to see if the attempt will be successful (i.e., there was an access pending and a legal Fastdata area address was used). Downloads will also shift in the data to be used to satisfy the load from the dmseg segment Fastdata area, while uploads will shift out the data being stored to the dmseg segment Fastdata area.

As noted above, two conditions must be true for the Fastdata access to succeed. These are:

- PrAcc must be 1, i.e., there must be a pending processor access.
- The Fastdata operation must use a valid Fastdata area address in the dmseg segment (0xF..F20.0000 to 0xF..F20.000F).

Table 7.12 shows the values of the PrAcc and SPrAcc bits and the results of a Fastdata access.

Table 7.12 Operation of the FASTDATA access

Probe Operation	Address Match check	PrAcc in the Control Register	LSB (SPrAcc) shifted in	Action in the Data Register	PrAcc changes to	LSB shifted out	Data shifted out
Download using FAST-DATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	write data	0 (SPrAcc)	1	valid (previous) data
		0	x	none	unchanged	0	invalid
Upload using FASTDATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	read data	0 (SPrAcc)	1	valid data
		0	x	none	unchanged	0	invalid

There is no restriction on the contents of the Data register. It is expected that the transfer size is negotiated between the download/upload transfer code that initiates the Fastdata access on the core and the probe software. To download a series of data words, the transfer code on the processor side would execute a loop of loads from the Fastdata memory area, and stores to target memory (accompanied by address increments). Note that the most efficient transfer sizes are word and double-word for 32-bit and 64-bit processors respectively.

The Rocc bit of the Control register is not used for the FASTDATA operation.

7.5.7 PCsample Register (PCSAMPLE Instruction)

Compliance Level: Required if PC Sampling feature is implemented in EJTAG (PC Sampling was introduced in EJTAG revision 3.xx.)

The PCSAMPLE instruction reads out the entire PCsample register. The width of the register depends on whether or not the processor implements the MIPS MT ASE. If MIPS MT is not implemented, the length is 41 bits. If MIPS MT is implemented, then the PCsample register length is 49 bits.

Figure 7.10 shows the format of the PCsample register; Table 7.8 describes the PCsample register field.

Figure 7.13 PCsample Register Format

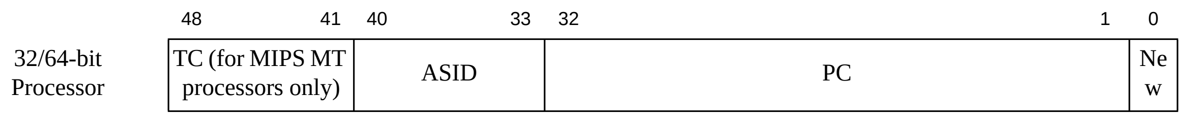


Table 7.13 PCsample Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
TC	48:41	Thread Context Id of the sampled PC.	R	Undefined	Required if MIPS MT is implemented
ASID	40:33	Address Space Id of the sampled PC	R	Undefined	Required
PC	32:1	Program Counter value	R	Undefined	Required
New	0	Processor writes a 1 to this field whenever a new sample is written into this register. The probe replaces with a zero when it reads out the sample value. Used to detect a duplicate sample read on the probe side.	R/W0	Undefined	Required

7.5.8 Bypass Register (TAP Instruction BYPASS, (EJTAG/NORMAL)BOOT, or Unused)

Compliance Level: Required with EJTAG TAP.

The Bypass register is a one-bit read-only register, which provides a minimum shift path through the TAP. This register is also defined in IEEE 1149.1.

Figure 7.14 shows the format of the Bypass register; Table 7.14 describes the Bypass register field.

Figure 7.14 Bypass Register Format

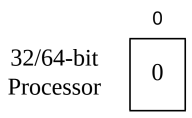


Table 7.14 Bypass Register Field Description

Fields		Description	Read / Write	Power-up State	Compliance
Name	Bits				
0	0	Ignored on writes; returns zero on reads.	R	0	Required

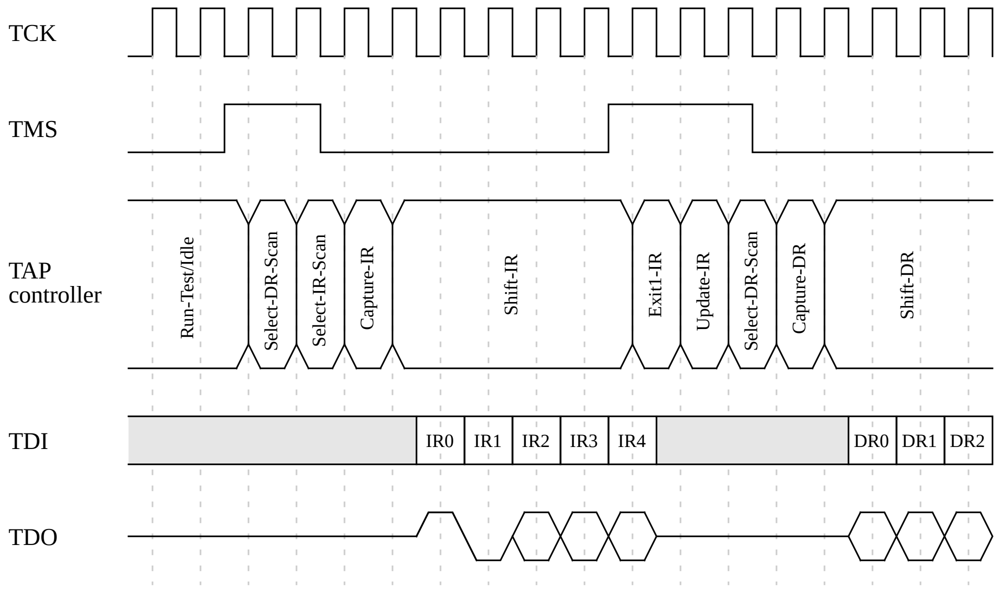
7.6 Examples of Use

This section provides several examples that use the TAP.

7.6.1 TAP Operation

An example for operation of the TAP is shown in Figure 7.15. TRST* is assumed deasserted high.

Figure 7.15 TAP Operation Example



The five-bit Instruction register is initially loaded with 00001_2. The first bit shifted out of the Instruction register is a 1 followed by four 0's. IR0 to IR4 indicate the new value for the Instruction register. IR0, the new LSB, is shifted in first, because it will be at the LSB position once all five bits are shifted in.

This example is similar for the selected data register.

7.6.2 ManufID Value

Table 7.15 shows the values of the ManufID field in the Device ID register as defined by the manufacturers. The Device ID register is described in Section 7.5.1 on page 127.

Table 7.15 ManufID Field Value Examples

Company	JEDEC Code	Continuations	Last byte without Carry	ManufID Value
Philips	0x15	0	0x15	0x015
LSI Logic	0xB6	0	0x36	0x036
IDT	0xB3	0	0x33	0x033
Toshiba	0x98	0	0x18	0x018
MIPS Technologies, Inc.	0x7F 7F A7	2	0x27	0x127

7.6.3 Rocc Bit Usage

The R/W0 Rocc bit in the EJTAG Control register acknowledges that the probe has seen a processor reset, and further accesses take this reset into account. This bit is set at reset. The probe must clear it as an acknowledge of the reset.

All other writes to the EJTAG Control register, except for the reset acknowledge, should write 1 to this bit in order to not acknowledge any resets occurring between reads and writes of the EJTAG Control register.

Correct use of the Rocc bit ensures safe handling of processor access even across reset. An example is the following scenario:

1. A processor access is pending and the PrAcc is read with value 1 (Rocc has been cleared previously).
2. The Address and Data registers are accessed and set up to handle the processor access.
3. The EJTAG Control register is accessed to finish the processor access. The register is read in the Capture-DR state. Shifting in of the value to write begins.
4. A reset of the processor occurs, the Rocc bit is set, and the PrAcc bit is cleared.
5. A new processor access occurs, because EJTAGBOOT was indicated.
6. A write of the EJTAG Control register is attempted with PrAcc equal to 0 and Rocc equal to 1, but the write does not occur because the Rocc bit is set. The new processor access that was not seen is not finished.
7. Polling of the EJTAG Control register continues. The probe detects that the Rocc bit is set.
8. The probe writes the EJTAG Control register with Rocc equal to 0 to acknowledge that the probe has seen the reset.
9. The new processor access is serviced as usual.

Inhibiting writes to the EJTAG Control register because of the Rocc bit ensures that the new processor access is not finished by mistake due to detection of a pending processor access before the reset occurred.

7.6.4 EJTAG Memory Access Through Processor Access

The processor access feature makes it possible for the probe to handle accesses from the processors to the specific EJTAG memory area (dmseg). Thus the processor can execute a debug handler from EJTAG memory, whereby applications that are not prepared with EJTAG code in the system memory still can be debugged.

The probe can get information about the access through the TAP as shown in Table 7.16.

Table 7.16 Information Provided to Probe at Processor Access

Information	Field and Register
Pending processor access	PrAcc field in the EJTAG Control register
Read or write access	PRnW field in the EJTAG Control register
Size and data location	Psz field in EJTAG Control register, and two or three LSBs in the Address register
Address	Address register
Data	Data register

The servicing of processor accesses works with a polling scheme, where the PrAcc bit is polled until a pending processor access is indicated by PrAcc equal to 1. Then the Address register is read to get the address of the transaction, and the Data register is accessed to get the write data or provide the read data. Finally the PrAcc bit is cleared, in order to finish the access from the processor.

In addition, the ProbTrap and ProbEn bits control the debug exception vector location and the indication to the processor that the probe will service accesses to the EJTAG memory through processor accesses.

Handling of processor access in relation to reset requires specific handling. A pending processor access is cleared at reset. At the same time, the Rocc bit is set, thereby inhibiting any processor accesses to be finished until Rocc is cleared. Thus the probe will have to acknowledge that a reset occurred, and will thereby not accidentally finish a processor access due to a processor access that occurred before the reset.

A pending processor access can only finish if the probe clears PrAcc or a processor reset occurs.

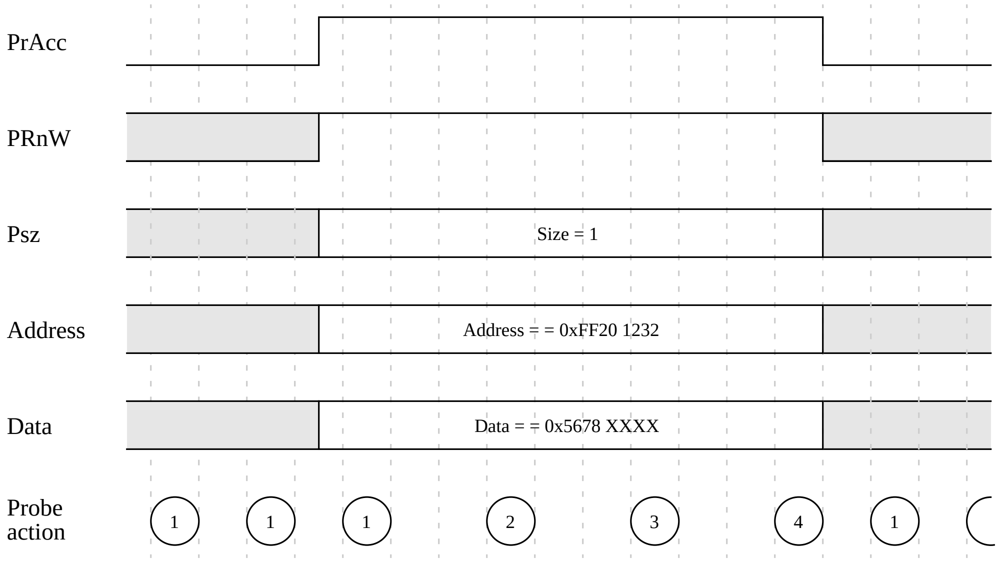
The width of the Address register is from 32 to 64 bits. The specific length is determined by shifting a known bit pattern through the register.

The following subsections show examples of servicing read and write processor accesses.

7.6.4.1 Write Processor Access

Figure 7.16 shows a possible flow for servicing a write processor access. The example implements a 32-bit processor with 32-bit Address register, running in little-endian mode. A halfword store is performed to address 0xFF20 1232 of value 0x5678.

Figure 7.16 Write Processor Access Example



The different probe actions shown on the figure are described below:

1. The EJTAG Control register is polled to get the indication for a pending PrAcc bit. The PrAcc bit is attempted to be written to 1 when polling, in order to prevent a processor access from finishing before being serviced. The values of PRnW and Psz are saved when PrAcc indicates a pending processor access.

2. The Address register is read. It contains the address of the store resulting in the write processor access.

3. The Data register is read, which contains the data from the store resulting in the write processor access.

4. The PrAcc bit is written to 0, in order to finish the processor access.

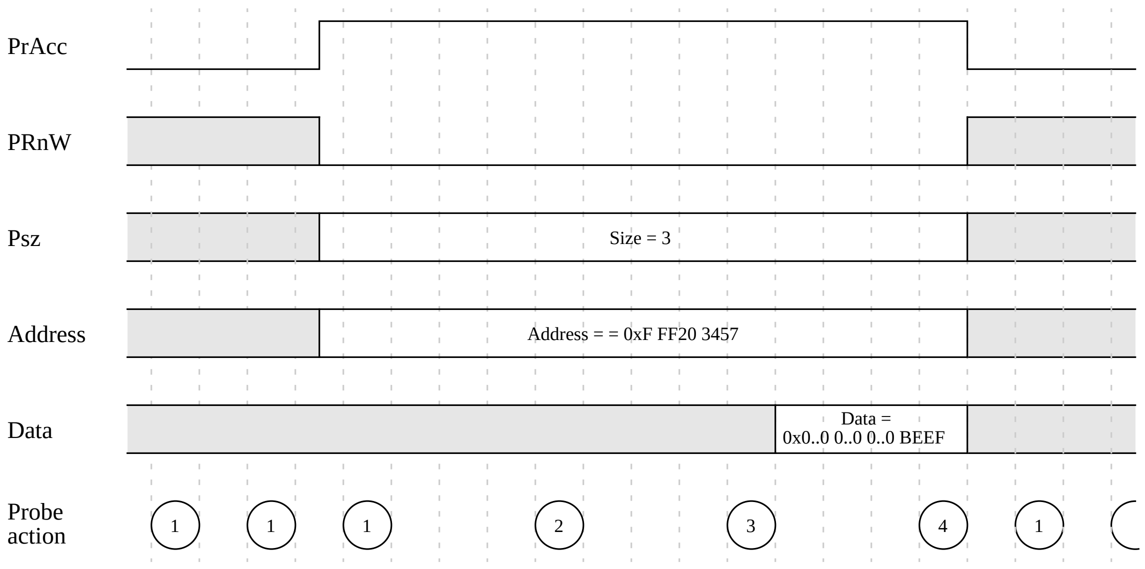
The probe must update the appropriate bytes in its internal memory used for EJTAG memory with the value of the write.

Notice that the two lower bytes of the Data register are undefined, and that the two lower bytes of the saved register are shifted up on the two high bytes in the Data register as on a data bus for an external memory system. The Address register in this case contains the address from the store; however, for some accesses, this is not the case because the two LSBs (32-bit processor) are modified for some accesses depending on size and address.

7.6.4.2 Read Processor Access

Figure 7.17 shows a possible flow for servicing a read processor access. The example implements a 64-bit processor with 36-bit Address register. A doubleword load/fetch from address 0xFFFF FFFF FF20 3450 is shown in the figure.

Figure 7.17 Read Processor Access Example



The different probe actions shown in the figure are described below:

1. The EJTAG Control register is polled to get the indication for a pending PrAcc bit. The PrAcc bit is attempted to be written to 1 when polling, in order to prevent a processor access from finishing before being serviced. The values of PRnW and Psz are saved when PrAcc indicates a pending processor access.

2. The Address register is read. It contains the address of the load/fetch resulting in the write processor access, with the three LSBs (64-bit processor) modified to allow size indication together with the Psz.

3. The Data register is written with the data to return for the load/fetch, resulting in the read processor access.

4. The PrAcc bit is cleared, in order to finish the processor access.

The probe must provide data for the read processor access from the internal EJTAG memory.

Notice that the Address register does not contain the direct address from the access, because the three LSBs (64-bit processor) are modified to indicate the size in conjunction with Psz. Also notice that in this case, there is no shifting of the data returned for the processor access by writing to the Data register, because a doubleword is provided. For other accesses, the Data register must be written with a shifted value depending on the specific access.

Chapter 8

On-Chip Interfaces

This chapter covers issues regarding implementation of a processor on a chip with respect to hook-up of the EJTAG TAP and DINT interfaces. It contains the following sections:

8.1 Connecting Unused EJTAG Test Access Port and Debug Interrupt Signals

If the EJTAG capabilities provided through the Test Access Port (TAP) and Debug Interrupt (DINT) signals on a processor core are unused when the processor core is implemented on a chip, then TRST* is tied to low (if TRST* is present on the core) and the remaining input signals TCK, TMS, TDI, and DINT must be tied to a constant value, either high or low. The output signal TDO should be left unconnected.

8.2 Optional TRST* Pin

The TRST* signal to the TAP is optional, and need not be provided as a pin on the chip for a processor implementing the EJTAG TAP.

If a TRST* chip pin is not provided, then a TAP reset like the one provided when TRST* is asserted (low) must be applied to the TAP at power-up, for example, through a power-up reset circuit on the chip. This power-up TAP reset must be finished after the time $T_{VIOrise}$ (see Section 9.2.4 on page 155).

If a TRST* chip pin is provided, then the power-up TAP reset is applied by a pull-down resistor, because the probe will not drive TRST* at power-up.

8.3 Input Buffers with Pull-Up/Down and Output Drivers for Chip Pins

If an input buffer with an integrated pull-up resistor is used for the TRST* chip pin, then its resistor value must be sufficiently large that it is overruled by the external pull-down resistor on the PCB, so a well-defined logical level is present on the TRST* pin (see Section 9.5.1 on page 157 for more information).

Observe the additional rules described in the IEEE Std. 1149.1 specification, if the same TAP is used for JTAG boundary scan also.

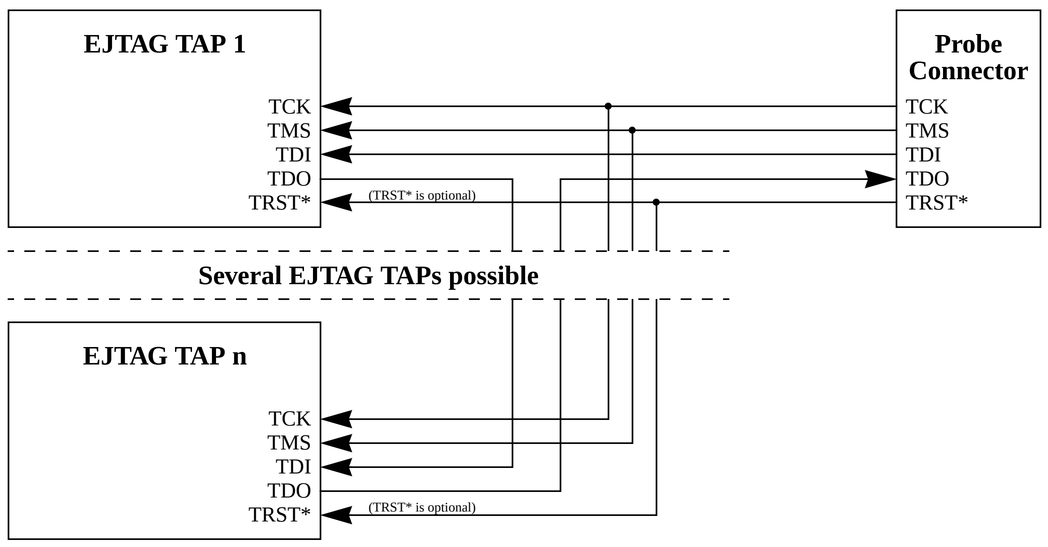
The output driver for the TDO chip pin must be capable of supplying the I_{OL} and I_{OH} current required for the probe (see Section 9.3 on page 155).

8.4 Connecting Multi-Core Test Access Port (TAP) Controllers

This section is concerned with building a multi-core system where each core has its own TAP controller, but share one set of external EJTAG TAP controller pins. Note that this section does not attempt to address the full issue of multi-core debug, which involves resolving debugger issues and other hardware issues such as debug signalling among multiple cores, and handling breakpoints across multiple cores, etc.

Figure 8.1 shows the recommended daisy-chain connection for a multi-core configuration, where the TCK, TMS and optional TRST* signals of all the TAP controllers are connected together. The TDI and TDO signals are daisy chained together so that the information flow between the selected register of all the TAP controllers is a continuous sequence.

Figure 8.1 Daisy-chaining of Multi-core EJTAG TAP Controllers



The simplest usage model for this multi-core connection, under most circumstance, only uses one “active” device. This is accomplished by including BYPASS TAP instruction for “non-active” devices in every TAP command chain sent by the debugger. “Non-active” devices only get attention when made “active”. Note that it is not necessary that only one device be “active” at a time, it depends entirely on how the debugger and the end-user want to control the multiple on-chip TAP controllers.

It is recommended that the EJTAG TAPs are connected in a single daisy-chain without any non-EJTAG TAPs in that chain, since this provide the fastest access to the EJTAG TAPs and it allows the most debug software packages to operate the EJTAG TAPs. Special care must be taken by the system designer if both EJTAG TAPs and non-EJTAG TAPs are connected in the same chain. In this case the system designer must ensure that both the EJTAG debug hardware and software, and the external device using the non-EJTAG TAPs can apply the BYPASS TAP instruction when the TAPs unrelated to the current operation are to be made “non-active”.

Chapter 9

Off-Chip and Probe Interfaces

This chapter outlines the requirements for the target system chip and probe interfaces to make them compatible. This chapter contains the following sections:

The off-chip interface forms the connection from the chip over the target system PCB and to the probe connector, thereby allowing the probe to connect to the target processor. The probe connection is optional in the target system.

The probe signals are described with respect to logical functionality, timing behavior, electrical characteristics, and connector and PCB design. Comments are also added with respect to probe functionality.

The descriptions in this chapter only cover issues related to EJTAG use of the Test Access Port (TAP). Issues related to reuse of the same TAP on a chip, for example, for JTAG boundary scan, are not covered.

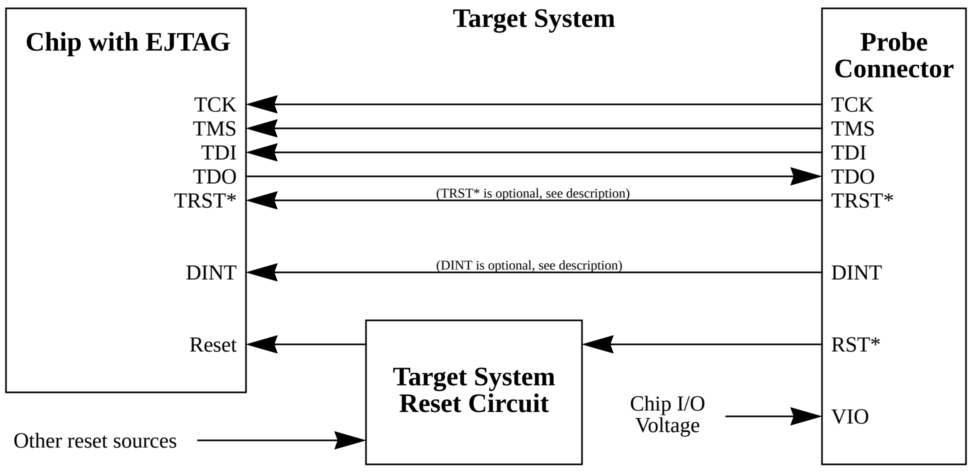
9.1 Logical Signals

This section describes the EJTAG signals categorized according to functionality:

- Test Access Port: TCK, TMS, TDI, TDO, and TRST* (optional TRST*)
- Debug Interrupt: DINT (optional)
- System reset (reset or soft reset): RST*
- Return TCK: RTCK (optional)
- Voltage Sense for I/O: VIO

Figure 9.1 shows the signal flow between the chip, target system PCB, and Probe.

Figure 9.1 Signal Flow Between Chip, Target System PCB, and Probe



9.1.1 Test Access Port Signals

The TCK, TMS, TDI, TDO, and TRST* signals make up the Test Access Port (TAP). For more details about the logical functionality of these signals, refer to Chapter 7, “EJTAG Test Access Port” on page 119. The five signals are listed in Table 9.1 with a short description.

Table 9.1 Test Access Port Signals Overview

Signal	Description	Direction	Compliance
TCK	Test Clock Input is the clock that controls the updates of the TAP controller and the shifts through the Instruction or selected data register(s). Both the rising and the falling edges of TCK are used.	Input	Required with probe connection
TMS	Test Mode Select Input is the control signal for the TAP controller. This signal is sampled at the rising edge of TCK.	Input	
TDI	Test Data Input has the data shifted into the Instruction or data register. This signal is sampled on the rising edge of TCK.	Input	
TDO	Test Data Output has the data shifted out from the Instruction or data register. This signal is changed on the falling edge of TCK.	Output	
TRST*	Test Reset Input is used for the TAP reset of the TAP controller, Instruction register, and EJTAGBOOT indication. TAP reset is applied asynchronously when low.	Input	Optional with probe connection

The TRST* chip pin is optional. If TRST* is not provided, then the TAP controller must be reset by a power-up reset circuit on-chip. Refer to Section 8.2 on page 147 for information on a power-up reset that is on-chip and Section 9.2.4 on page 155 for duration of this power-up reset.

9.1.2 Debug Interrupt Signal

The Debug Interrupt (DINT) signal allows the probe to request the CPU to take a debug exception. Table 9.2 briefly defines this signal.

Table 9.2 Debug Interrupt Signal Overview

Signal	Description	Direction	Compliance
DINT	A debug interrupt is requested when DINT goes from low to high. The CPU is allowed to synchronize this signal to the CPU clock before detecting its rising edge, if this is possible with respect to the minimum pulse width indicated in Section 9.2.2 on page 154. The request is ignored if the CPU is already in Debug Mode.	Input	Optional with EJTAG TAP

The DINT signal from the probe is optional. The DINTsup bit indicates whether or not the DINT signal is implemented. Refer to Section 7.5.2 on page 128 for more information on DINTsup. The debug interrupt request is described in Section 6.3.9 on page 98.

9.1.3 System Reset Signal

The System Reset (RST*) signal from the probe is required to generate a reset of the target board. It is recommended that assertion of RST* results in a (hard) reset of the processor, but it is allowed to generate a soft reset. Table 9.3 briefly describes the RST* signal.

Table 9.3 System Reset Signal Overview

Signal	Description	Direction	Compliance
RST*	RST* is the system reset of the target board. When the probe asserts RST* low, the result is either a reset (recommended) or soft reset of the processor. No reset is applied when the RST* is undriven (3-stated from the probe).	Input	Required with probe connection

The probe controls the RST* via an open-collector (OC) output. Thus RST* is actively driven low when asserted (low), but is 3-stated when deasserted (high).

9.1.4 Return Test Clock Input

The Voltage sense for I/O (VIO) indicates target power is applied and voltage levels are present at the probe I/O connections. Table 9.5 briefly describes the VIO signal.

Table 9.4 Voltage Sense for I/O Signal Overview

Signal	Description	Direction	Compliance
RTCK	This return TCK signal to the JTAG connector allows the target chip under debug to mirror the start and stop of its system clock to correspond to start and stop of the debug probe.	Input	Optional with probe connection

This is useful when for example, a hardware emulator used with the target core wants to hook up an EJTAG probe for debugging. The hardware emulator starts and stops its system clock and needs the debug probe to pause any JTAG operations during that time. This can be achieved by the addition of a return TCK signal which is an output from the target chip to the probe and is a mirror of the probe's TCK input after clocking with the system clock. The probe can be configured in a mode where it will wait for RTCK to be equal to TCK before proceeding with the scan. This would then allow the JTAG port to be throttled by the parget as needed.

9.1.5 Voltage Sense for I/O Signal

The Voltage sense for I/O (VIO) indicates target power is applied and voltage levels are present at the probe I/O connections. Table 9.5 briefly describes the VIO signal.

Table 9.5 Voltage Sense for I/O Signal Overview

Signal	Description	Direction	Compliance
VIO	Voltage Sense for I/O indicates if target power is applied, and indicates the voltage level for the probe signals.	Output	Required with probe connection

With VIO, the probe can auto adjust the voltage level for the signals, and detect if power is lost at the target system.

9.2 AC Timing Characteristics

The timing relations and AC requirements for the signals are described in this section. The timing is measured at the probe connector for the target system, and must be valid in the full operating range of the target board.

All setup and hold times are measured with respect to the 50% value between V_{IL} / V_{IH} for inputs, and V_{OL} / V_{OH} for outputs.

All rise and fall times are measured at 20% and 80% of the values of V_{IL} / V_{IH} for inputs and V_{OL} / V_{OH} for outputs.

The capacitance of C_{Target} and C_{Probe} is assumed to be as seen from the probe connector for the inputs and outputs.

9.2.1 Test Access Port Timing

Figure 9.2 shows the timing relationships of the five TAP signals, TCK, TMS, TDI, TDO, and TRST*. Table 9.6 shows the absolute times for the symbols in the figure.

Figure 9.2 Test Access Port Signals Timing

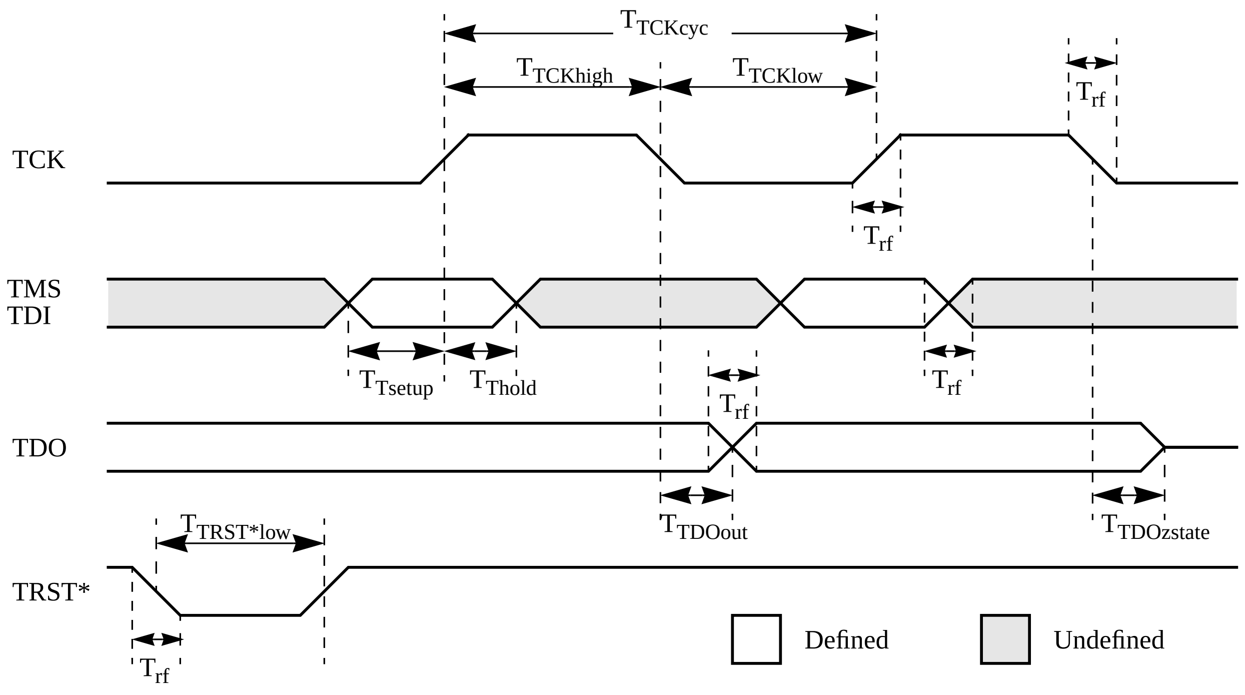


Table 9.6 Test Access Port Signals Timing Values

Symbol	Description	Min	Max	Unit
T_{TCKcyc}	TCK cycle time	25		ns
$T_{TCKhigh}$	TCK high time	10		ns
T_{TCKlow}	TCK low time	10		ns
T_{Tsetup}	TAP signals setup time before rising TCK	5		ns
T_{Thold}	TAP signals hold time after rising TCK	3		ns
T_{TDOout}	TDO output delay time from falling TCK		5	ns
$T_{TDOzstate}$	TDO 3-state delay time from falling TCK		5	ns
$T_{TRST*low}$	TRST* low time	25		ns
T_{rf}	TAP signals rise / fall time, all input and output		3	ns

TRST* is independent of the TCK signal, because TRST* is a truly asynchronous signal. Note the IEEE 1149.1 recommendation in 3.6.1 (d): “To ensure deterministic operation of the test logic, TMS should be held at 1 while the signal applied at TRST* changes from 0 to 1.” A race might otherwise occur if TRST* is deasserted (going from low to high) on a rising edge of TCK when TMS is low, because the TAP controller might go either to Run-Test/Idle state or stay in the Test-Logic-Reset state.

9.2.2 Debug Interrupt Timing

Figure 9.3 shows the timing for the DINT signal from the probe. Table 9.7 shows the absolute times for the symbols in the figure.

Figure 9.3 Debug Interrupt Signal Timing

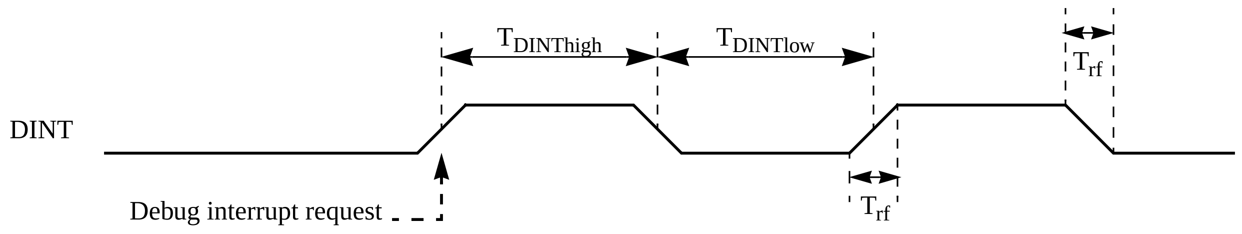


Table 9.7 Debug Interrupt Signal Timing Values

Symbol	Description	Min	Max	Unit
$T_{DINThigh}$	DINT high time	1		µs
$T_{DINTlow}$	DINT low time	1		µs
T_{rf}	DINT signal rise / fall times		3	ns

The probe should guarantee that the $T_{DINThigh}$ and $T_{DINTlow}$ pulse widths meet the specifications, in order to leave enough time for the CPU to synchronize the DINT signal to the internal CPU clock domain.

If the CPU clock speed or clocking scheme is such that $T_{DINThigh}$ and $T_{DINTlow}$ do not leave enough time for synchronization or, for example, PLL walk-up, then the target system is responsible for extending the DINT pulse in the processor.

9.2.3 System Reset Timing

Figure 9.4 shows the timing for the RST* signal from the probe. Table 9.8 shows the absolute times for the symbols in the figure. The target system is responsible for extending the RST* pulse if required.

Figure 9.4 System Reset Signal Timing

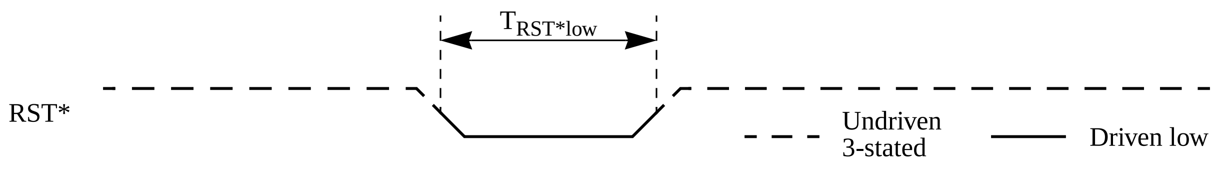


Table 9.8 System Reset Signal Timing Value

Symbol	Description	Min	Max	Unit
$T_{RST*low}$	RST* low time	1		ms

9.2.4 Voltage Sense for I/O (VIO) Timing

Figure 9.5 shows the timing for the VIO signal. Table 9.9 shows the absolute time for the symbol in the figure. VIO must rise to the stable level within a specific time $T_{VIOrise}$ after the probe detects VIO to be above a certain limit $V_{VIOactive}$.

Figure 9.5 Voltage Sense for I/O Signal Timing

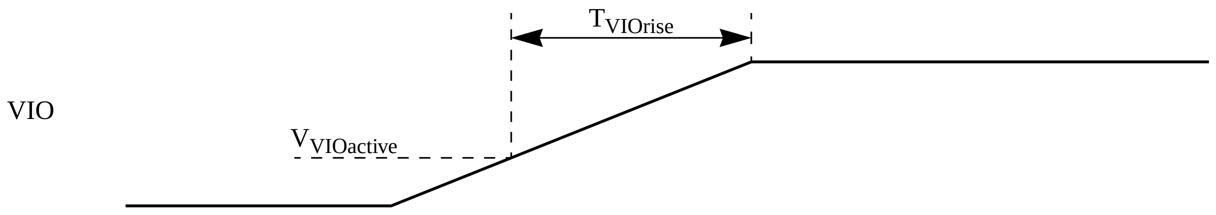


Table 9.9 Voltage Sense for I/O Signal Timing Value

Symbol	Description	Min	Max	Unit
$T_{VIOrise}$	VIO rise time from $V_{VIOactive}$ to stable VIO value		2	s

The target system must ensure that $T_{VIOrise}$ is obeyed after the $V_{VIOactive}$ value is reached, so the probe can use this value to determine when the target has powered-up. The probe is allowed to measure the $T_{VIOrise}$ time from a higher value than $V_{VIOactive}$ (but lower than V_{VIO} minimum) because the stable indication in this case comes later than the time when target power is guaranteed to be stable.

If TRST* is asserted by a pulse at power-up, either on-chip or on PCB, then this reset must be completed after $T_{VIOrise}$. If TRST* is asserted by a pull-down resistor, then the probe will control TRST*.

At power-down no power is indicated to the probe when VIO drops under the $V_{VIOactive}$ value, which the probe uses to stop driving the input signals, except for RST*.

9.3 DC Electrical Characteristics

Table 9.10 describes the DC electrical characteristics for voltage and current measured at the probe connector. Current measures positive in direction from the probe to the target system, and negative in the other direction. The characteristics apply to the full operating range of the target system.

Table 9.10 DC Electrical Characteristics

Symbol	Description	Condition	Min	Typ	Max	Unit
V_{VIO}	VIO voltage	When stable	1.5		5.0	V
$V_{VIOactive}$	VIO active indication			0.5		V
I_{VIO}	VIO output current				20	mA

Table 9.10 DC Electrical Characteristics (Continued)

Symbol	Description	Condition	Min	Typ	Max	Unit
V_{IL}	Low-level input voltage	$2.8\ V \le V_{VIO}$	- 0.3		0.8	V
		$V_{VIO} < 2.8\ V$	- 0.3		$0.3 * V_{VIO}$	V
V_{IH}	High-level input voltage	$2.8\ V \le V_{VIO}$	2.0		$V_{VIO} + 0.3$	V
		$V_{VIO} < 2.8\ V$	$0.7 * V_{VIO}$		$V_{VIO} + 0.3$	V
V_{OL}	Low-level output voltage	$2.8\ V \le V_{VIO}$	- 0.3		0.4	V
		$V_{VIO} < 2.8\ V$	- 0.3		$0.15 * V_{VIO}$	V
V_{OH}	High-level output voltage	$2.8\ V \le V_{VIO}$	2.4		$V_{VIO} + 0.3$	V
		$V_{VIO} < 2.8\ V$	$0.85 * V_{VIO}$		$V_{VIO} + 0.3$	V
I_{IL}	Low-level input current, except RST*		- 8.0			mA
I_{RST}	RST* low-level input current		- 10			mA
I_{IH}	High-level input current				8.0	mA
I_{OL}	Low-level output current				8.0	mA
I_{OH}	High-level output current		- 8.0			mA
I_{Zstate}	3-state input or output current	$0\ V \le V_{sig} \le V_{VIO}$	- 50		50	µA
C_{Target}	Capacitance for target system				25	pF
C_{Probe}	Capacitance for probe				25	pF

The I_{Zstate} specifies the current that a 3-stated (undriven) output driver and pull-up/down can provide. It sets a limit for the drivers in the probe for TCK, TMS, TDI, TRST*, DINT, and RST*, and it sets a limit for the output driver on-chip for TDO. This limit allows design of pull-up/down resistors that can keep a logical level when no driver is controlling the signal.

C_{Target} and C_{Probe} are the capacitances in the target system for inputs and the capacitances for the probe for outputs. Additional capacitance in the target system must be added to C_{Probe} when designing the output driver, and additional capacitance for the probe driver is added to C_{Target}.

9.4 Mechanical Connector

Figure 9.6 shows the recommended EJTAG connector on a target system. The connector is a common pin strip with dimensions 0.100” x 0.100”, for example, SAMTEC part number TSW-107-23-L-D or compatible. The socket on the probe side must allow for an angled connector on the target system.

Figure 9.6 EJTAG Connector Mechanical Dimensions

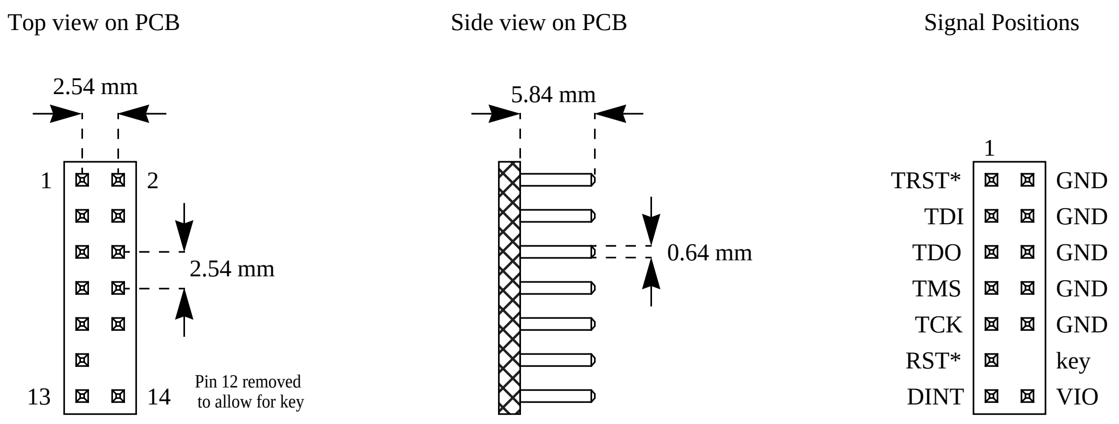


Table 9.11 shows the pin assignments for the connector.

Table 9.11 EJTAG Connector Pinout

Pin	Signal	Direction	Pin	Signal	Direction
1	TRST* - Test Reset Input	Input	2	GND - Ground	GND
3	TDI - Test Data Input	Input	4	GND - Ground	GND
5	TDO - Test Data Output	Output	6	GND - Ground	GND
7	TMS - Test Mode Select Input	Input	8	GND - Ground	GND
9	TCK - Test Clock Input	Input	10	GND - Ground	GND
11	RST* - System Reset	Input	12	RTCK - Return Test Clock Input	Input
13	DINT - Debug Interrupt	Input	14	VIO - Voltage Sense for I/O	Output

With older EJTAG connectors, Pin 12 on the target system connector should be removed to provide keying and thereby ensure correct connection of the probe to the target system. But with the enhancement with the RTCK signal, generation of RTCK is indicated by the presence of pin 12 on the target connector.

The connector in Figure 9.6 does not provide PC trace signals. An additional connector, probably with 0.05” x 0.05” spacing, will be defined later when the PC trace feature is redefined.

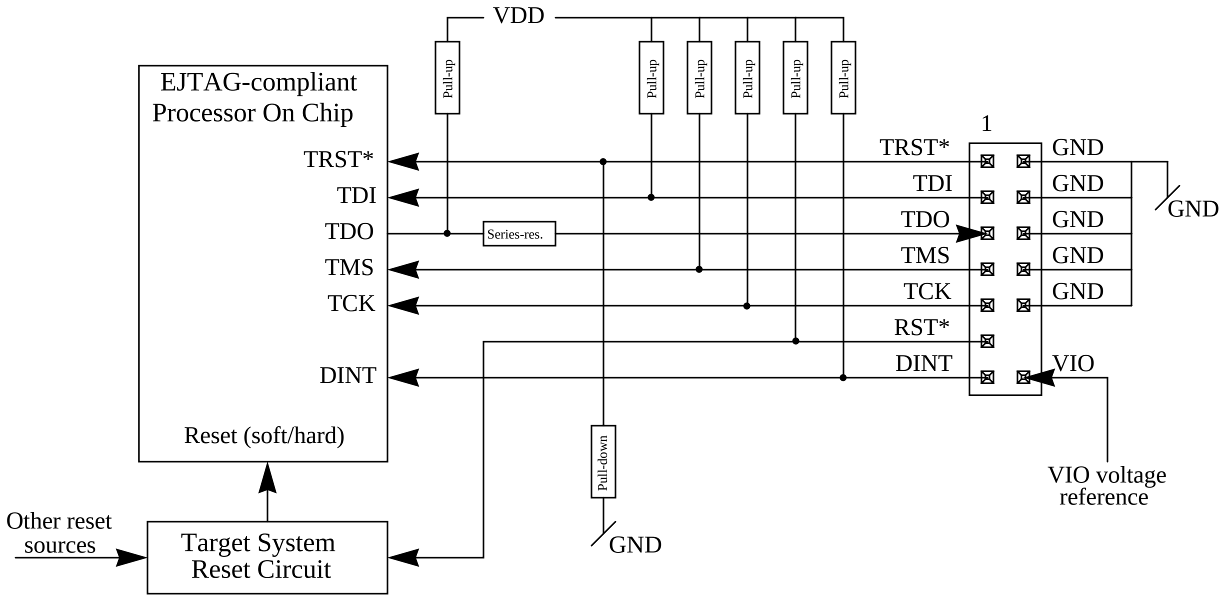
9.5 Target System PCB Design

This section provides guidelines for using the EJTAG connector on a target system.

9.5.1 Electrical Connection

Figure 9.7 shows the electrical connection of the target system connector. This subsection only covers the case where the probe connects directly to a chip with an EJTAG compliant processor.

Figure 9.7 Target System Electrical EJTAG Connection



In Figure 9.7, the pull-up resistors for TCK, TMS, TDI, DINT, and RST*, the pull-down resistor for TRST*, and the series resistor for TDO must be adjusted to the specific design. However, the recommended pull-up/down resistor is 1.0 kΩ, because a low value reduces crosstalk on the cable to the connector, allowing higher TCK frequencies. A typical value for the series resistor is 33 Ω. Recommended resistor values have 5% tolerance.

The IEEE 1149.1 specification requires that the TAP controller is reset at power-up, which can occur through a pull-down resistor on TRST* if the probe is not connected. However, on-chip pull-up resistors can be implemented on some chips due to an IEEE 1149.1 requirement. Having on-chip pull-up and external pull-down resistors for the TRST* signal requires special care in the design to ensure that a valid logical level is provided to TRST*, for example, using a small external TRST* pull-down resistor to ensure this level overrides the on-chip pull-up. An alternative is to use an active power-up reset circuit for TRST*, which drives TRST* low only at power-up and then holds TRST* high afterwards with a pull-up resistor.

It must be ensured that a valid logical level is provided on TRST*, because some chips have an on-chip pull-down resistor on TRST* (even through this setup contradicts the IEEE 1149.1 standard), which might cause an undefined signal value when other chips have on-chip pull-ups, and they all connect to TRST*.

The pull-up resistor on TDO must ensure that the TDO level is high when no probe is connected and the TDO output is 3-stated. This requirement allows reliable connection of the probe if it is hooked-up when the power is already on (hot plug). The value of the pull-up resistor depends on the 3-state current of the TDO output driver in the chip, but a value around 47 kΩ usually is sufficient.

Optional diodes to protect against overshoot and undershoot voltage can be provided on the signals to the chip with EJTAG.

The RST* signal must have a pull-up resistor because it is controlled by an open-collector (OC) driver in the probe, and thus is actively pulled low only. The pull-up resistor is responsible for the high value when not driven by the probe. The input on the target system reset circuit must be able to accept the rise time when the pull-up resistor charges the C_{Target} and C_{Probe} capacitance to a high logical level.

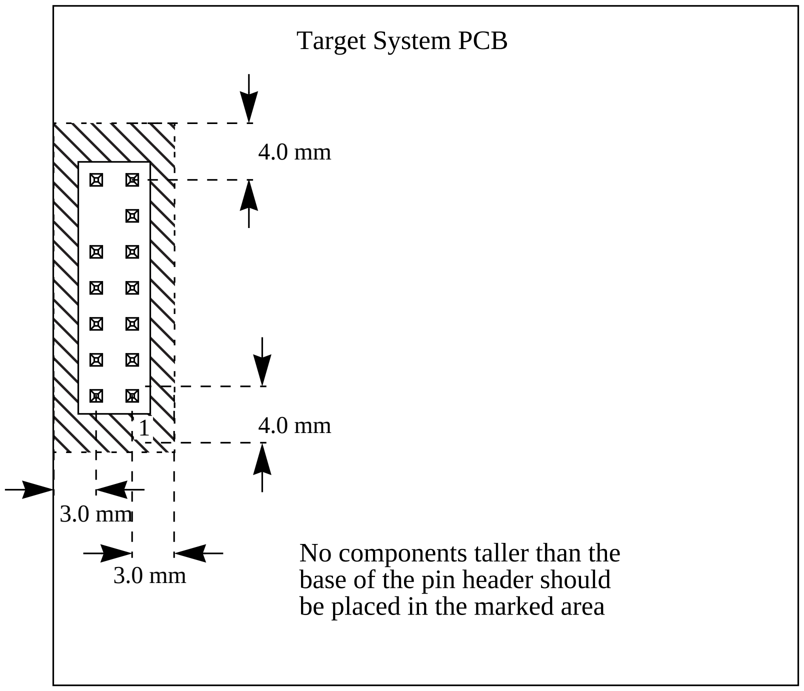
VIO must connect to a voltage reference that drops rapidly to below $V_{VIOactive}$ when the target system loses power, even with the capacitive load of C_{Probe}. The probe can thus detect the lost power condition.

The signals on the probe connection for the optional signals DINT and TRST* should be left unconnected in the target system, if unused.

9.5.2 Layout Considerations

Layout around the pin connector on the target system must provide for sufficient clearance for the probe to connect. Figure 9.8 shows the recommended clearance. Place the connector at the edge of the PCB. Avoid tall components around the connector to allow for easy access.

Figure 9.8 Target System Layout for EJTAG Connection


Target System PCB
4.0 mm
4.0 mm
1
3.0 mm
3.0 mm
No components taller than the base of the pin header should be placed in the marked area

9.6 Probe Requirements and Recommendations

This section provides the probe requirements for different features.

9.6.1 Target System Power-Up with Probe Attached

A probe connected to the target system at power-up is not allowed to drive the inputs before VIO indicates a stable voltage (see Section 9.2.4 on page 155). TRST* (if present) is then asserted by the target system pull-down resistor at power-up, whereby a TAP reset is applied through TRST* for TAPs, depending on TRST*. This step implies that inputs are not driven until the target system is powered up; otherwise the communication on the TAP might be undefined or damage could occur.

At power-down the probe is not allowed to drive the inputs after VIO has dropped under a certain level (see Section 9.2.4 on page 155).

The RST* signal is an exception to the above description because it can be driven low by the probe during power-up.

9.6.2 Hot Plug in of Probe

The probe must not drive any inputs to the target system if it is connected while the system is running (hot plug). Detection of a stable VIO from the target system is required before any input is allowed to be (see Section 9.2.4 on page 155).

To avoid spikes or changes in the input voltage to the target system when the probe is connected, the level of the signal on the probe must be adjusted to the same level as the signals on the target system. This adjustment can be done with large pull-up/down resistors (in the range of 150 kΩ) on the probe signals, so the level of these signals matches the level on the target system shown in Figure 9.8. The specific implementation of this feature is dependent on the probe, the driver type, etc. used in the probe.

9.6.3 TDO Level when 3-Stated

The probe must apply a pull-up resistor on TDO to have a well-defined logical level when TDO on the TAP is 3-stated. The pull-up on the target system ensures the level at hot plug. The size of the pull-up on the probe is expected to be 1.0 kΩ or more. The resistor value must be chosen so I_{Zstate} is observed.

9.6.4 RST* Drive by Open Collector

Drive the RST* signal with an open-collector (OC) output driver to allow for easy connection of the RST* signal in the target system.

9.6.5 Changing TMS and TDI

It is recommended that the TMS and TDI signals driven by the probe change in relation to the falling edge generated on the TCK, since this ensures a high setup and hold time for the TMS and TDI in relation to the rising edge of TCK, on which these signals are sampled by the target processor.

If the TCK clock speed can be adjusted by extending the high and low period time of the TCK clock, then the behavior described above will also make the probe work even with a target processor not respecting setup and hold time, simply by lowering the TCK frequency.

9.6.6 Mechanical Connector

The female connector from the probe must allow for an angled board connector.

Block Hole 12 on the probe connector in order to provide keying and ensure correct connection of the probe to the target system. Connect the signal from the probe at line 12 to GND on the probe.

With the enhancement of the EJTAG connector with the input RTCK signal on pin 12, targets generating RTCK can only be used with probes capable of accepting it. Generation of RTCK is indicated by the presence of pin 12 on the target connector. Probe acceptance of RTCK is indicated by lack of a plug on pin 12 of the probe cable.

Appendix A

Differences for R3k Privileged Environments

This appendix describes the EJTAG feature differences necessary for integration with a 32-bit processor having an R3k privileged environment.

A.1 EJTAG Processor Core Extensions

This section covers differences between an R3K environment and the description in Chapter 6, “EJTAG Processor Core Extensions” on page 81.

A.1.1 SYNC Instruction

The SYNC instruction is not available for processors with R3k privileged environment, but this instruction must be available and have behavior as described in Section 6.2.3.7 on page 88.

A.1.2 Debug Exception Vector Location

Table A.1 shows the debug exception vector location in system memory for processors with R3k privileged environments.

Table A.1 Debug Exception Vector Location for R3k Privileged Environment Processors

ProbTrap bit in ECR register	Debug Exception Vector Address
0	0xBFC0 0200

The debug exception vector in dmseg (EJTAG memory) is the same for processors with R3k and R4k privileged environments.

A.1.3 SYNC Instruction Substitute

In case the SYNC instruction is not provided (for example, on a processor with an R3k privileged environment), then an implementation-specific instruction sequence must be used to ensure full update of the Debug register status bits and BSn bits for hardware breakpoints with respect to handling of imprecise data hardware breakpoints and imprecise errors.

A.1.4 CP0 Register Numbers for Debug and DEPC Registers

The register numbers to use in processors with R3k privileged environments for CP0 Debug and DEPC registers is shown below:

- Debug register: 16
- DEPC register: 17

A.2 Hardware Breakpoints

This section describes the differences between hardware breakpoints in an R3k privileged environment and those describes in Chapter 3, “Hardware Breakpoints” on page 33.

A.2.1 Instruction Breakpoint Registers

Table A.2 shows the address offsets in drseg for the Instruction Breakpoint registers. In the table, n is the breakpoint number in the range 0 to 14.

Table A.2 Offsets for Instruction Breakpoint Registers for R3k Privileged Environment Processors

Offset in drseg	Register Mnemonic	Register Name and Description
0x0004	IBS	Instruction Breakpoint Status
0x0100 + 0x010 * n	IBAn	Instruction Breakpoint Address n
0x0104 + 0x010 * n	IBCn	Instruction Breakpoint Control and ASID n
0x0108 + 0x010 * n	IBMn	Instruction Breakpoint Address Mask n

A.2.2 Conditions for Matching Instruction Breakpoints

The width in bits of the ASID field for the compare is 6 bits, as is the size used in the TLB. The ASID and $IBASIDn_{ASID}$ references used in the equations in Section 3.3.1 on page 36 has this size.

A.2.3 ASID Field in IBCn Register

Compliance Level: Required with instruction breakpoints when the ASIDsup bit in the IBS register is 1, optional otherwise.

The ASID field has the ASID value used in the compare for instruction breakpoint n; it is placed in the IBCn register, not in a register of its own. Table A.3 shows the format of the ASID field.

Table A.3 ASID Field in IBCn Register

Fields		Description	Read/ Write	Reset State
Name	Bits			
ASID	29:24	Instruction breakpoint ASID value for compare.	R/W	Undefined

A.2.4 Data Breakpoint Registers

Table A.4 shows the address offsets in drseg for the Data Breakpoint registers. In the table, n is the breakpoint number in the range 0 to 14.

Table A.4 Offsets for Data Breakpoint Registers for R3k Privileged Environment Processors

Offset in drseg	Register Mnemonic	Register Name and Description
0x0008	DBS	Data Breakpoint Status

Table A.4 Offsets for Data Breakpoint Registers for R3k Privileged Environment Processors (Continued)

Offset in drseg	Register Mnemonic	Register Name and Description
0x0200 + 0x010 * n	DBAn	Data Breakpoint Address n
0x0204 + 0x010 * n	DBCn	Data Breakpoint Control and ASID n
0x0208 + 0x010 * n	DBMn	Data Breakpoint Address Mask n
0x020C + 0x010 * n	DBVn	Data Breakpoint Value n

A.2.5 Conditions for Matching Data Breakpoints

The width in bits of the ASID field for the compare is 6 bits, as is the size used in the TLB. The ASID and $\text{DBASIDn}_{\text{ASID}}$ references used in the equations in Section 3.3.2 on page 38 has this size.

A.2.6 ASID Field in DBCn Register

Compliance Level: Required with instruction breakpoints when the ASIDsup bit in the DBS register is 1, optional otherwise.

The ASID field has the ASID value used in the compare for data breakpoint n; it is placed in the DBCn register, not in a register of its own. Table A.5 shows the format of the ASID field.

Table A.5 ASID Field in DBCn Register

Fields				
Name	Bits	Description	Read/ Write	Reset State
ASID	29:24	Data breakpoint ASID value for compare.	R/W	Undefined

A.3 EJTAG Test Access Port

There are no differences for processors with R3k privileged environment with respect to the EJTAG Test Access Port. The R4k/R3k bit in the Implementation register selects between R4k and R3k privileged environments (see Section 7.5.2 on page 128).

Appendix B

Terminology

This appendix defines several terms used throughout this document.

Term	Definition
3-state	Undriven output, thus output with high impedance
ASE	Application Specific Extension.
CP0	Coprocessor 0 (zero)
Debug exception	Exception bringing the processor from Non-Debug Mode to Debug Mode.
Debug Mode exception	Exception occurring in Debug Mode, which causes the processor to re-enter Debug Mode.
dmseg	Memory-mapped area, accessible from the processor in Debug Mode only. It is provided as emulated memory handled by the probe through processor accesses.
drseg	Memory mapped area, accessible from the processor in Debug Mode only. It contains registers for hardware breakpoint setup, for example.
dseg	Memory mapped area, accessible from the processor in Debug Mode only. It contains the combined dmseg and drseg areas.
EJTAG	Enhanced JTAG.
EJTAG Area	See dseg definition.
EJTAG Memory	See dmseg definition.
EJTAG Registers	See drseg definition.
GPR	General-Purpose Registers r0 to r31.
IEEE 1149.1	IEEE standard describing the TAP and the boundary-scan architecture.
ISA	Instruction Set Architecture.
JTAG	Joint Test Action Group.
Hardware breakpoint	Instruction or data breakpoints implemented in hardware.
LSB	Least Significant Bit.
MMU	Memory Management Unit. Translates virtual addresses to physical addresses.
MSB	Most Significant Bit.
Naturally-aligned	Alignment of a memory structure at an address corresponding to its size, so for example a word is aligned to an word boundary thus where the two LSBs of the address are 0.
Non-Debug Mode	Any mode other than Debug Mode (User Mode, Supervisor Mode or Kernel Mode).
PC	Program Counter, the virtual address of the currently executed instruction.
Probe	A hardware system controlling the target system through the TAP. The probe is controlled through the debug host, a PC, or workstation.
Processor access	Access from the processor to dmseg, which is handled by the probe through the TAP.

Term	Definition
Software breakpoint	SDBBP instruction, which can be inserted in the code being debugged, causing a debug exception when executed.
TAP	Test Access Port. The interface port defined in IEEE 1149.1 and used for access to EJTAG from the probe. The interface is made up of the test clock (TCK), test mode select (TMS), test data in (TDI), test data out (TDO), and optional TAP reset (TRST*).
TLB	Translation Lookaside Buffer. Provides programmable mapping of address translations done by the MMU.
Triggerpoint	Hardware breakpoint, which is set up to generate a trigger indication when it matches.

Appendix C

Functional Clarifications from Old EJTAG 2.5

The following items were clarified from the previous EJTAG rev. 2.5 Specification:

- Update of Instruction register in Update-IR state

  Updating Instruction register in the Update-IR state is allowed either on the rising or the falling TCK edge. See Section 7.3.4 on page 123 for more information.

- Update of selected Data register(s) in Update-DR state

  Updating selected Data register(s) in the Update-IR state is allowed either on the rising or the falling TCK edge. See Section 7.3.7 on page 123 for more information.

- Use of the Device ID register

  The Device ID register is recommended to be unique among designs and among several processors on the same chip. See Section 7.5.1 on page 127 for more information.

- Reset State or Power-up State

  Either the reset state or the power-up state is indicated for the data registers. It is not possible to state only the reset value, because a reset denotes a processor reset. For example, the Bypass register must be reset to 1 as soon as the TAP can be operated, thus the processor should not be required to be reset first. See Section 7.5 on page 126 for more information.

- SRstE Changed to Optional

  The SRstE bit described in Chapter 2, “Debug Control Register” on page 29 has been made optional, because not every implementation needs it, and its behavior is defined as implementation dependent.

- Bypass Register Initial Value as 0 (zero)

  The initial value for the Bypass register (in Capture-DR state) is defined to 0 (zero), see Section 7.5.8 on page 141., since the JTAG spec. requires this in chapter 9 page 9-1.

Appendix D

Multithreaded and Multi-Core Debug

This is not a required feature of EJTAG, but is provided here has a recommended method to implement debug for a multi-core or a multi-threaded processor.

D.1 Introduction

This document serves as a guideline for designing a Multi-Core Breakpoint Unit (MCBU) for System-On-Chip (SOC) devices that integrate multiple MIPS processor cores. The document is intended to be used by designers of SOC devices and by software tool vendors who design debuggers capable of interacting with these SOC devices.

The MCBU is capable of requesting a debug interrupt from any number of cores in the SOC as a result of any core in the system entering Debug Mode. In addition, the MCBU can be used to request debug interrupt, soft reset, hard reset and non-maskable interrupt from any number of the cores under software control.

D.2 MCBU Register Map

The MCBU consists of registers that specify which of the processors in the multi-processor system should receive a RESET, COLD RESET, NMI, and Debug Interrupt signal. There are also per-processor debug interrupt registers that say whether that processor would cause a debug interrupt to be sent to other processors in the multi-processor system. These registers are described below. These registers are memory-mapped for access by the debug probe hardware and software and the memory map is shown in Table D.1 and Table D.2.

Table D.1 sMCBU Register Memory Map

Register Name	Memory Map of the Register
Reset	Base+0x000
Cold_Reset	Base+0x010
NMI	Base+0x020
Debug_Interrupt	Base+0x030

Table D.2 MCBU Debug_Int Register Memory Map

Register Name	Memory Map of the Register
Debug_Int_0	Base+0x200
Debug_Int_1	Base+0x210
Debug_Int_2	Base+0x220
...	...

Table D.2 MCBU Debug_Int Register Memory Map

Register Name	Memory Map of the Register
Debug_Int_i	Base+0x200+(0x10*i_{16}), (i expressed in hex)
...	...
Debug_Int_63	Base+0x5F0

SoC designers are advised to design the base address to be 0x1FFFC00. This is the end of kseg1 (ROM is at 0x1FC00000). If it is impossible to map the MCDU into this address, SoC designers are requested to map base into kseg1 and to notify the head of the Architecture Team at MIPS Technologies of the selected base address. Debugger designers are advised to use the above-specified address as the default, but to enable configuring this address in the debuggers for SoC devices that are using a different address. A default configuration file (mips_mcbu_base.cfg) could be made available by the chip manufacturer to the debugger vendors.

Addresses Base through Base+0x1FFF should be reserved for future expansion of the MCBU. If no more than N cores are implemented in the SoC (N < 32), only registers Debug_Int_0 through Debug_Int_N-1 need to be implemented. Registers Debug_Int_N through Debug_Int_31 should remain reserved.

D.3 MCBU Registers

D.3.1 Debug_Int_i

There are a maximum of 64 such registers, but only as many as exist in the multiprocessor system needs to be implemented. The Debug_Int_i register is a 64-bit read/write register that contains a mask used to control which of the processor cores in the SOC device should receive an EJ_DINT request upon a detection of an asserted EJ_DebugM in processor core number “i” in the SOC. When Mask[j] is set, an asserted EJ_DebugM in processor core number “i” will force the EJ_DINT in core number “j” to be asserted. When Mask[j] is clear, an asserted EJ_DebugM in processor core number “i” will have no effect on EJ_DINT in core number “j”.

If no more than N cores are implemented in the SOC (N < 64), bits N through 63 should remain reserved. Upon SOC reset, the value of the Mask bits is undefined.

Figure D.1 Debug_Int_i Register Format

63 k+1	k 1 0
0	Mask

Table D.3 Debug_Int_i Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Power-up State**	**Compliance**
Mask	k:0	There are k+1 processors in the multi-processor system under debug. For each processor, the corresponding mask bit, that is, mask[j] for processor j specifies whether or not the current processor i will assert EJ_DINT for j when i gets a EJ_DebugM.	R/W	0	Required if MCBU is implemented
0	63:k+1	Reserved	R	0	Required if MCBU is implemented

D.3.2 Reset

The Reset register is a 64-bit read/write register that contains a mask used to control which of the processor cores in the SoC device should receive a SI_Reset request. When Mask[j] is set, the MCDU will force the SI_Reset input of core “j” to be asserted.

If no more than N cores are implemented in the SoC (N < 64), bits N through 63 should remain reserved. Upon SoC reset, the value of the Mask bits is undefined.

Figure D.2 Reset Register Format

63 k+1	k 1 0
0	Mask

Table D.4 Reset Register Field Descriptions

Fields		Description	Read / Write	Power-up State	Compliance
Name	Bits				
Mask	k:0	There are k+1 processors in the multi-processor system under debug. When the mask bit j is set, this forces a SI_Reset signal to processor j.	R/W	0	Required if MCBU is implemented
0	63:k+1	Reserved	R	0	Required if MCBU is implemented

D.3.2.1 Cold Reset

The Cold Reset register is a 64-bit read/write register that contains a mask used to control which of the processor cores in the SoC device should receive a SI_ColdReset request. When Mask[j] is set, the MCDU will force the SI_ColdReset input of core “j” to be asserted.

If no more than N cores are implemented in the SoC (N < 64), bits N through 63 should remain reserved. Upon SoC reset, the value of the Mask bits is undefined.

Figure D.3 Cold Reset Register Format

63 k+1	k 1 0
0	Mask

Table D.5 Cold Reset Register Field Descriptions

Fields		Description	Read / Write	Power-up State	Compliance
Name	Bits				
Mask	k:0	There are k+1 processors in the multi-processor system under debug. When the mask bit j is set, this forces a SI_ColdReset signal to processor j.	R/W	0	Required if MCBU is implemented
0	63:k+1	Reserved	R	0	Required if MCBU is implemented

D.3.2.2 NMI

The NMI register is a 64-bit read/write register that contains a mask used to control which of the processor cores in the SoC device should receive a SI_NMI request. When Mask[j] is set, the MCDU will force the SI_NMI input of core “j” to be asserted.

If no more than N cores are implemented in the SoC (N < 64), bits N through 63 should remain reserved. Upon SoC reset, the value of the Mask bits is undefined.

Figure D.4 NMI Register Format

63 k+1	k 1 0
0	Mask

Table D.6 NMI Register Field Descriptions

Fields		Description	Read / Write	Power-up State	Compliance
Name	Bits				
Mask	k:0	There are k+1 processors in the multi-processor system under debug. When the mask bit j is set, this forces a SI_NMI signal to processor j.	R/W	0	Required if MCBU is implemented
0	63:k+1	Reserved	R	0	Required if MCBU is implemented

D.3.3 Debug Interrupt

The Debug Interrupt register is a 64-bit read/write register that contains a mask used to control which of the processor cores in the SoC device should receive a EJ_DINT request. When Mask[j] is set, the MCDU will force the EJ_DINT input of core “j” to be asserted.

If no more than N cores are implemented in the SoC (N < 64), bits N through 63 should remain reserved. Upon SoC reset, the value of the Mask bits is undefined.

Figure D.5 Debug Interrupt Register Format

63 k+1	k 1 0
0	Mask

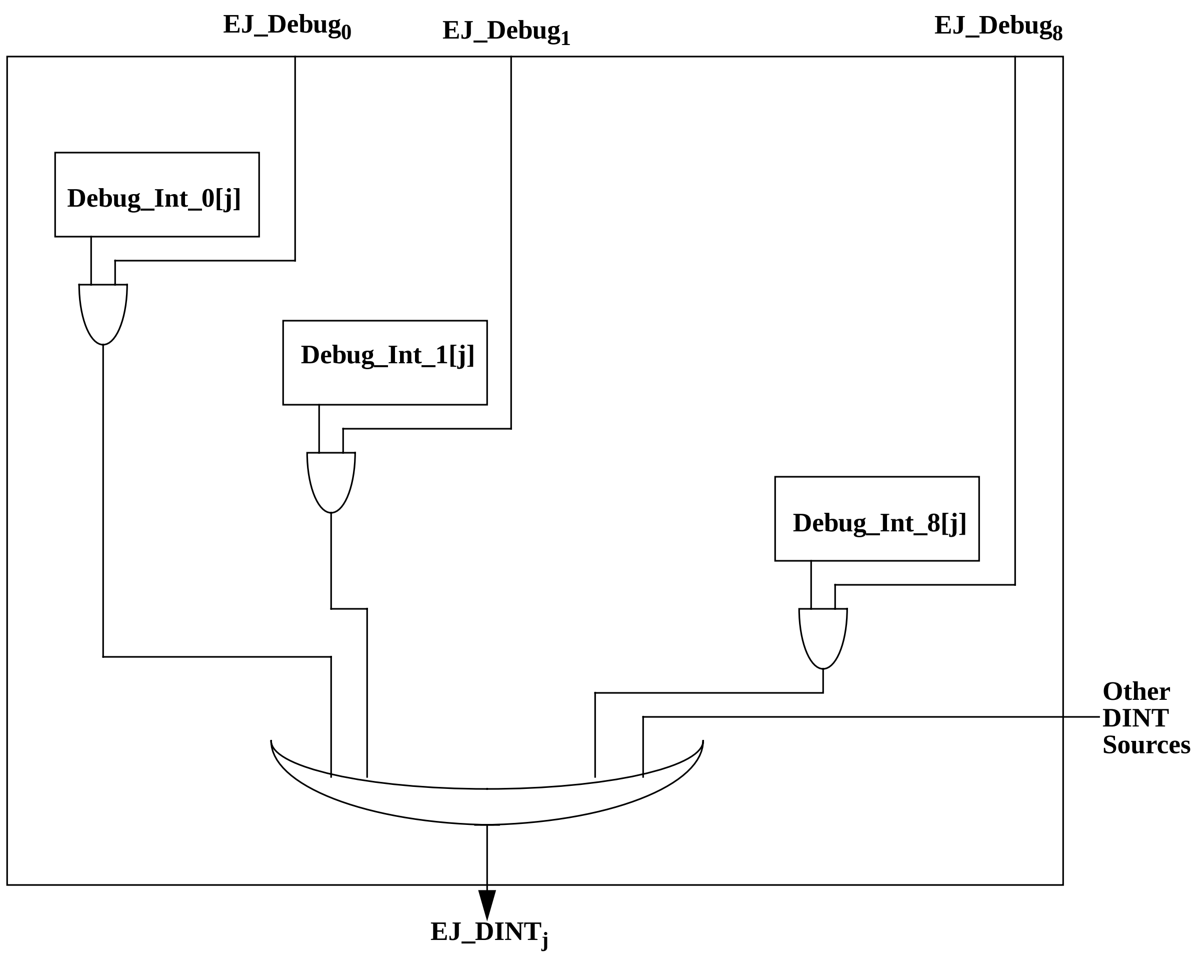
Table D.7 Debug Interrupt Register Field Descriptions

Fields		Description	Read / Write	Power-up State	Compliance
Name	Bits				
Mask	k:0	There are k+1 processors in the multi-processor system under debug. When the mask bit j is set, this forces a EJ_DINT signal to processor j.	R/W	0	Required if MCBU is implemented
0	63:k+1	Reserved	R	0	Required if MCBU is implemented

D.4 Possible Implementation

The following diagram demonstrates a possible implementation of a circuit that generates EJ_DINT to processor "j" in a system with 9 processors

Figure D.6 An Example Implementation



Appendix E

DRSEG Memory Map

This appendix lists the various registers mapped into the debug register segment (drseg).

Offset	Register	Section Reference
0x00000	Debug Control Register	Chapter 2, “Debug Control Register” on page 29
0x00004	*Instruction Breakpoint Status Register (Old)*	*Section A.2.1 “Instruction Breakpoint Registers”*
0x00008	*Data Breakpoint Status Register (Old)*	*Section A.2.4 “Data Breakpoint Registers”*
0x00100-0x001FF	*Instruction Breakpoint Control Registers (Old)*	*Section A.2.1 “Instruction Breakpoint Registers”*
0x00200-0x002FF	*Data Breakpoint Control Registers (Old)*	*Section A.2.4 “Data Breakpoint Registers”*
0x01000	Instruction Breakpoint Status	Section 3.6.1 “Instruction Breakpoint Status (IBS) Register”
0x01100-0x01FE0	Instruction Breakpoint Control (15 breakpoints)	Section 3.6.2 “Instruction Breakpoint Address n (IBAn) Register” - Section 3.6.5 “Instruction Breakpoint Control n (IBCn) Register”
0x01FF8	TraceIBPC2 Register	The PDtrace™ Interface and Trace Control Block Specification (MD00439)
0x02000	Data Breakpoint Status (New)	Section 3.7.1 “Data Breakpoint Status (DBS) Register”
0x02100-0x02FE0	Data Breakpoint Control (15 breakpoints)	Section 3.7.2 “Data Breakpoint Address n (DBAn) Register” - Section 3.7.5 “Data Breakpoint Control n (DBCn) Register”
0x02FF0	Load Data Value Register	Section 3.3.3 “Precise Exceptions on Data Value Match Breaks”
0x02FF8	TraceDBPC2 Register	The PDtrace™ Interface and Trace Control Block Specification (MD00439) Revision 6.00 (or newer)
0x3000	TCBControlA	
0x3008	TCBControlB	
0x3010	TCBControlC	
0x3018	TCBControlD	
0x3020	TCBControlE	
0x3028	TCBConfig	
0x03100	TCBTW	
0x03108	TCBRDP	
0x03110	TCBWRP	
0x03118	TCBSTP	
0x03120	BKUPRDP	
0x03128	PKUPWRP	
0x03130	BKUPSTP	
0x3200-0x3238	TCBTrigX	

Offset	Register	Section Reference
0x03FC0	iFlowTCB Control/Status Register	The iFlowtrace™ Architecture Specification (MD00526)
0x03FC8	iFlowTrace Write Address Register	
0x03FD0	ITrigiFlowTrcEn Register	
0x03FD8	DTrigiFlowTrcEn Register	
0x04000-0x07FFF	On chip SRAM or Trace Memory (iFlowTrace)	
0x08000	Complex Break and Trigger Control Register	Section 4.3.1 “Complex Break and Trigger Control (CBTC) Register (0x8000)”
0x08300-0x084DF	PrCndAI[n], n=0..14	Section 4.7 “Primed Breakpoints”
0x084E0-0x086BF	PrCndAD[n], n=0..14	
0x08900	Stopwatch Timer Control	Section 4.3.7 “Stopwatch Timer Control (STCtl) Register (0x8900)”
0x08908	Stopwatch Timer Count	Section 4.3.8 “Stopwatch Timer Count (STCnt) Register (0x8908)”

Appendix F

Revision History

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself. Certain parts of this document (Instruction set descriptions, EJTAG register definitions) are references to Architecture specifications, and the change bars within these sections indicate alterations since the previous version of the relevant Architecture document.

Revision	Date	Description
2.5	February 22, 2000	Release to users under NDA
2.5-1	June 6, 2000	Changes in this revision: • Clarification describing possible speculative fetch from dmseg. See Section 6.2.2.1 on page 85. • Clarification of SYNC instruction behavior in Section 6.2.3.7 on page 88. • Added hazard description on DEBUG[LSNM] and DEBUG[IEXI] in Section 6.2.4 on page 89. • Clarification for Doze and Halt bits in Debug register, see Section 6.7.1 on page 104. • Removed requirement that bytes of TAP Data Register which are not accessed for a processor access read must be written with 0s by the probe. Thus, now any value may be written to the not accessed bytes. • Wording change in headline and beginning of Appendix C covering clarification of changes since previous EJTAG revisions. • Added cross references for clarification. • Corrected typos. • Declassify the document.
2.5-2	August 22, 2000	Removed old Section 6.2, and added Section 6.4 to discuss multi-core EJTAG, i.e., MIPS recommended way to connect multiple TAP controllers to one set of external EJTAG TAP pins.
02.53	January 8, 2001	Changes in this revision: • Revision number changed to have format XX.YY, thus the next minor revision after 2.5-2 is named 02.53. • Clarification of data triggerpoint handling when exception occur on a load/store instruction. • Clarification of value of BYTELANE for hardware breakpoints when access with unaligned address occurs. • Elaborated description of fields in TAP Device ID register. • Added recommendation for handling of CacheErr register in Debug Mode. • Modified description of connecting multiple TAP controllers in daisy chain. • Updates for clarifications in general. • Corrected typos.

Revision	Date	Description
02.60	February 15, 2001	Changes in this revision: • Updated the chapter on TAP controller to specify the FASTDATA instruction. • Added the instructions needed for the trace control block register access. • Updated the revision number to 02.60 and made a value of 2 in the EJTAGver field correspond to this version.
02.61	September 30, 2002	Changes in this revision: • Include the EJTAGver field encoding of 2, inadvertently left out of version 2.60.
02.62	May 7, 2003	Changes in this revision: • Remove Appendix D, as this information in not appropriate to a specification documenting the current state of the EJTAG architecture. • Clarify the definition of EJTAGBOOT. If this condition is active, the first instruction fetch after reset is to one of the EJTAG debug addresses, not to the reset exception vector. • Clarify the wording describing the BAI field of the Data Breakpoint Control register. • Clarify the definition of ADDR for the LUXC1 and SUXC1 instructions, when used in the data breakpoint address match equation. • Clarify the use of the $\text{Debug}_{\text{DExcCode}}$ field for SDBBP instructions in Debug Mode. • Add an introduction to EJTAG to the first chapter of the specification. • Clarify the state of the Halt and Doze bits in the Debug register if a hardware interrupt or other event awakens the processor, but a debug exception is taken instead. • Make it clear that it is implementation dependent whether an SC/SCD, which would fail because the LLbit is 0, will cause a debug exception due to a data breakpoint match. • Update with MIPS32 and MIPS64 Release 2 Architecture changes.
3.10	July 5, 2005	Changes in this revision: • Added PC Sampling feature • Added support for MIPS MT ASE • EJTAG version 3 for specification revision 3.10 and up • Inclusion of a possible proposal for implementing EJTAG support for multiple processors or a multi-threaded configuration • Miscellaneous cleanup
3.20	September 19, 2005	Changes in this revision: • PC sampling clarifications for MT, add a PCSe bit to DCR • Typo fixes
4.00	June 28, 2006	Changes in this revision: • Add complex break and trigger chapter and the Debug2 register • Add the ability to Invert a data value check • Add the feature that saves a data value on a precise match • Typo fixes and clarification.
4.10	July 3, 2006	Fix typographical errors, unresolved pointers, and clarification of existing features. Add a new Return TCk (RTCK) signal to pin 12 of the EJTAG Connector.
4.11	May 18, 2007	Add EJTAGJver 4.0 to indicate the architecture upgrade to include the Complex Break and Trigger feature.
4.12	July 15, 2008	Update copyrights and contact information.

Revision	Date	Description
4.13	August 01, 2008	• DBCCn Register figure was missing UnPRCnd field. • Load Data Value address offset is 0x2FF0. • Page 16, Table 1.1 and Page 138, 7.5.5.1 gave the wrong impression that EJTAGBOOT and NORMALBoot commands cause reset themselves.
4.14	November 06, 2008	• Added new TAP instructions • Added drseg map appendix