Chapter 13
Debug

"watchpoint" = fault was a load or store
"breakpoint" = fault was b/c address loaded by program counter.

13-5: cheat sheet of debug registers.
13-26: cheat sheet of assembly instructions (always: opcode_1=00, CRn=0)
13-34: effect on DFSR, IFSR, FAR, WFAR registers.
13-45: cookbook for setting up watchpoints / breakpoints.

This chapter describes the processor debug unit, that assists development of application software, operating systems, and hardware, and contains the following sections:

key facts:
- we are doing monitor-debug
- *cannot* do mismatch watchpoints in priviledged modes (all modes other than USER)

13.1 Debug systems

The processor forms one component of a debug system that interfaces from the high-level debugging performed by you, to the low-level interface supported by the processor. Figure 13-1 shows a typical system.

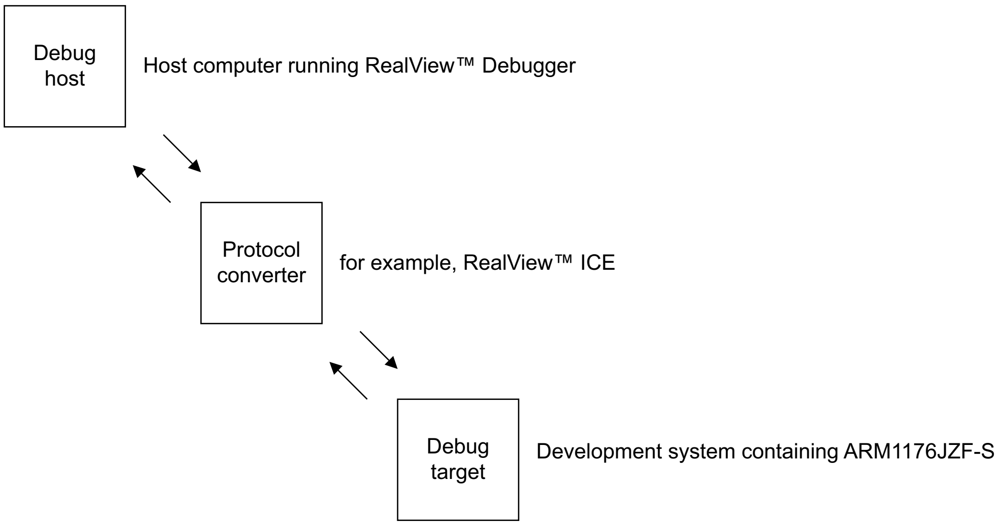
Debug host
Host computer running RealView™ Debugger
Protocol converter
for example, RealView™ ICE
Debug target
Development system containing ARM1176JZF-S

Figure 13-1 Typical debug system

This typical system has three parts:

- *The debug host*
- *The protocol converter*
- *The processor*.

13.1.1 The debug host

after this lab can build your own real debugger

The debug host is a computer, for example a personal computer, running a software debugger such as RealView Debugger. The debug host enables you to issue high-level commands such as *set breakpoint at location XX*, or *examine the contents of memory from 0x0-0x100*.

13.1.2 The protocol converter

The debug host is connected to the processor development system using an interface, for example an RS232. The messages broadcast over this connection must be converted to the interface signals of the processor. This function is performed by a protocol converter, for example, RealView ICE.

13.1.3 The processor

The processor, with debug unit, is the lowest level of the system. The debug extensions enable you to:

- stall program execution
- examine its internal state and the state of the memory system
- resume program execution.

The debug host and the protocol converter are system-dependent.

13.2 About the debug unit

The processor debug unit assists in debugging software running on the processor. You can use the processor debug unit, in combination with a software debugger program, to debug:

- application software
- operating systems
- ARM processor based hardware systems.

The debug unit enables you to:

- stop program execution
- examine and alter processor and coprocessor state
- examine and alter memory and input/output peripheral state
- restart the processor core.

You can debug the processor in the following ways:

- *Halting debug-mode debugging*
- *Monitor debug-mode debugging*
- Trace debugging. See Chapter 15 *Trace Interface Port* for interfacing with an ETM.

us

The processor debug interface is based on the *IEEE Standard Test Access Port and Boundary-Scan Architecture*.

~~13.2.1~~ Halting debug-mode debugging

skip everything with debug-mode. needs external system.

When the processor debug unit is in Halting debug-mode, the processor halts and enters Debug state when a debug event, such as a breakpoint, occurs. When the processor is in Debug state, an external host can examine and modify its state using the DBGTAP.

In Debug state you can examine and alter processor state, processor registers, coprocessor state, memory, and input/output locations through the DBGTAP. This mode is intentionally invasive to program execution. Halting debug-mode debugging requires:

- external hardware to control the DBGTAP
- a software debugger to provide the user interface to the debug hardware.

See *CP14 c1, Debug Status and Control Register (DSCR)* on page 13-7 to learn how to set the processor debug unit into Halting debug-mode.

13.2.2 Monitor debug-mode debugging

this is just a normal exception.

When the processor debug unit is in Monitor debug-mode, the processor takes a Debug exception instead of halting. A special piece of software, a debug monitor target, can then take control to examine or alter the processor state. Monitor debug-mode is essential in real-time systems where the core cannot be halted to collect information. For example, engine controllers and servo mechanisms in hard drive controllers that cannot stop the code without physically damaging the components.

When debugging in Monitor debug-mode the processor stops execution of the current program and starts execution of a debug monitor target. The state of the processor is preserved in the same manner as all ARM exceptions. See the *ARM Architecture Reference Manual* on exceptions and exception priorities. The debug monitor target communicates with the debugger to access processor and coprocessor state, and to access memory contents and input/output peripherals. Monitor debug-mode requires a debug monitor program to interface between the debug hardware and the software debugger.

When debugging in Monitor debug-mode, you can program new debug events through CP14. This coprocessor is the software interface of all the debug resources such as the breakpoint and watchpoint registers. See *CP14 c1, Debug Status and Control Register (DSCR)* on page 13-7 to learn how to set the processor debug unit into Monitor debug-mode.

Note

Monitor debug-mode, used for debugging, is not the same as Secure Monitor mode.

13.2.3 Secure Monitor mode and debug

Debug can be restricted to one of three levels, Non-secure only, Non-secure and Secure User only, or any Secure or Non-secure levels so that you can prevent access to Secure parts of the system while still permitting Non-secure and optionally Secure User parts to be debugged. This is controlled by the **SPIDEN** and **SPNIDEN** signals and the two bits SUIDEN and SUNIDEN in the Secure Debug Enable Register in the system control coprocessor, see *External debug interface* on page 13-28 and *c1, Secure Debug Enable Register* on page 3-54.

Invasive debug

Invasive debug is debug where the system can be both observed and controlled like all of the debug in this section that enables you to halt the processor and examine and modify registers and memory.

SPIDEN and SUIDEN control invasive debug permissions.

Non-invasive debug

Non-invasive is debug where the system can only be observed but not affected. The ETM interface, the System Performance Monitor and the DBGTAP program counter sample register provide non-invasive debug.

SPNIDEN and SUNIDEN control non-invasive debug permissions.

13.2.4 Virtual addresses and debug

confusing: talks about virtual addresses, but it *appears* we can use physical (I have been)

Unless otherwise stated, all addresses in this chapter are *Modified Virtual Addresses* (MVA) as the *ARM Architecture Reference Manual describes.* For example, the *Breakpoint Value Registers* (BVR) and *Watchpoint Value Registers* (WVR) must be programmed with MVAs.

The terms *Instruction Modified Virtual Address* (IMVA) and *Data Modified Virtual Address* (DMVA), where used, mean the MVA corresponding to an instruction address and the MVA corresponding to a data address respectively.

13.2.5 Programming the debug unit

The processor debug unit is programmed using *CoProcessor 14* (CP14). CP14 provides:

- instruction address comparators for triggering breakpoints
- data address comparators for triggering watchpoints
- a bidirectional *Debug Communication Channel* (DCC)
- all other state information associated with processor debug.

CP14 is accessed using coprocessor instructions in Monitor debug-mode, and certain debug scan chains in Debug state, see Chapter 14 *Debug Test Access Port* to learn how to access the processor debug unit using scan chains.

13.3 Debug registers

Table 13-1 lists definitions of terms used in register descriptions.

Table 13-1 Terms used in register descriptions

Term	Description
R	Read-only. Written values are ignored. However, it is written as 0 or preserved by writing the same value previously read from the same fields on the same processor.
W	Write-only. This bit cannot be read. Reads return an Unpredictable value.
RW	Read or write.
C	Cleared on read. This bit is cleared whenever the register is read.
UNP/SBZP	Unpredictable or *Should Be Zero or Preserved* (SBZP). A read to this bit returns an Unpredictable value. It is written as 0 or preserved by writing the same value previously read from the same fields on the same processor. These bits are usually reserved for future expansion.
Core view	This column defines the core access permission for a given bit.
External view	This column defines the DBGTAP debugger view of a given bit.
Read/write attributes	This is used when the core and the DBGTAP debugger view are the same.

in general, keep an eye out for this! can't read > 1.

On a power-on reset, all the CP14 debug registers take the values indicated by the Reset value column in the register bit field definition tables:

- Table 13-4 on page 13-8
- Table 13-6 on page 13-14
- Table 13-11 on page 13-18
- Table 13-14 on page 13-21
- Table 13-16 on page 13-21.

In these tables, - means an Undefined Reset value.

13.3.1 Accessing debug registers

To access the CP14 debug registers you must set Opcode_1 and CRn to 0. The Opcode_2 and CRm fields of the coprocessor instructions are used to encode the CP14 debug register number, where the register number is `{<Opcode2>, <CRm>}`.

Table 13-2 lists the CP14 debug register map. All of these registers are also accessible as scan chains from the DBGTAP.

Table 13-2 CP14 debug register map

cheat sheet of the different registers / their encoding.

Binary address		Register number	CP14 debug register name	Abbreviation
Opcode_2	CRm			
b000	b0000	c0	Debug ID Register	DIDR
b000	b0001	c1	Debug Status and Control Register	DSCR
b000	b0010-b0100	c2-c4	Reserved	-
b000	b0101	c5	Data Transfer Register	DTR

Table 13-2 CP14 debug register map (continued)

Binary address		Register number	CP14 debug register name	Abbreviation
Opcode_2	CRm			
b000	b0110	c6	Watchpoint Fault Address Register	WFAR
b000	b0111	c7	Vector Catch Register	VCR
b000	b1000-b1001	c8-c9	Reserved	-
b000	b1010	c10	Debug State Cache Control Register	DSCCR
b000	b1011	c11	Debug State MMU Control Register	DSMCR
b000	b1100-b1111	c12-c15	Reserved	-
b001-b011	b0000-b1111	c16-c63	Reserved	-
b100	b0000-b0101	c64-c69	Breakpoint Value Registers	BVRy[a]
	b0110-b111	c70-c79	Reserved	-
b101	b0000-b0101	c80-c85	Breakpoint Control Registers	BCRy[a]
	b0110-b1111	c86-c95	Reserved	-
b110	b0000-b0001	c96-c97	Watchpoint Value Registers	WVRy[a]
	b0010-b1111	c98-c111	Reserved	-
b111	b0000-b0001	c112-c113	Watchpoint Control Registers	WCRy[a]
	b0010-b1111	c114-c127	Reserved	-

[a] y is the decimal representation for the binary number CRm.

fault b/c
address loaded
into pc

fault b/c address
was loaded into
register using
ld/st. only 2
of these.

Note

All the debug resources required for Monitor debug-mode debugging are accessible through CP14 registers. For Halting debug-mode debugging some additional resources are required. See Chapter 14 *Debug Test Access Port*.

13.3.2 CP14 c0, Debug ID Register (DIDR)

The Debug ID Register is a read-only register that defines the configuration of debug registers in a system. Figure 13-2 shows the format of the Debug ID Register.

useful facts about the current
machine. (always the same
for us i believe since all
the A+'s are from the
same batch. curious if not
though)

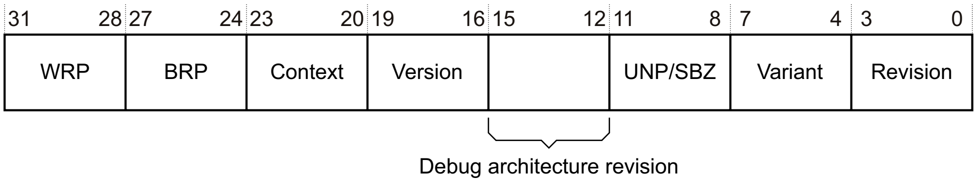

Figure 13-2 Debug ID Register format

For the ARM1176JZF-S processor:

- DIDR[31:8] has the value 0x15121x

- the value of DIDR[7:0] is determined by fields in the CP15 c0 Main ID Register, as described in the field descriptions in Table 13-3.

Table 13-3 lists the bit field definitions for the Debug ID Register.

Table 13-3 Debug ID Register bit field definition

Bits	Read/write attributes	Description

[31:28]
WRP
	R	Number of Watchpoint Register Pairs:
		b0000 = 1 WRP
		b0001 = 2 WRPs
		…
		b1111 = 16 WRPs.
		For the ARM1176JZF-S processor these bits are b0001 (2 WRPs).

[27: 24] BRP	R	Number of Breakpoint Register Pairs: b0000 = Reserved. The minimum number of BRPs is 2. b0001 = 2 BRPs b0010 = 3 BRPs … b1111 = 16 BRPs. For the ARM1176JZF-S processor these bits are b0101 (6 BRPs).

[23: 20]
Context
	R	Number of Breakpoint Register Pairs with context ID comparison capability:
		b0000 = 1 BRP has context ID comparison capability
		b0001 = 2 BRPs have context ID comparison capability
		…
		b1111 = 16 BRPs have context ID comparison capability.
		For the ARM1176JZF-S processor these bits are b0001 (2 BRPs).

[19:16] Version	R	Debug architecture version. `0x2` denotes v6.1
[15:12]	R	Debug architecture revision 0x1 denotes TrustZone features
[11:8]	UNP/SBZP	Reserved.
[7: 4] Variant	R	Implementation-defined variant number, incremented on major revisions of the product. This field is identical to bits [23:20] of the CP15 c0 Main ID Register, see *c0, Main ID Register* on page 3-20.
[3: 0] Revision	R	Implementation-defined revision number, incremented on minor revisions of the product. This field is identical to bits [3:0] of the CP15 c0 Main ID Register, see *c0, Main ID Register* on page 3-20.

The reason for duplicating the Variant and Revision fields here is that the Debug ID Register is accessible through scan chain 0. This enables an external debugger to determine the variant and revision numbers without stopping the core.

13.3.3 CP14 c1, Debug Status and Control Register (DSCR)

use to enable
debugging. without
this nothing will happen.

The Debug Status and Control Register contains status and configuration information about the state of the debug system. Figure 13-3 on page 13-8 shows the format of the Debug Status and Control Register.

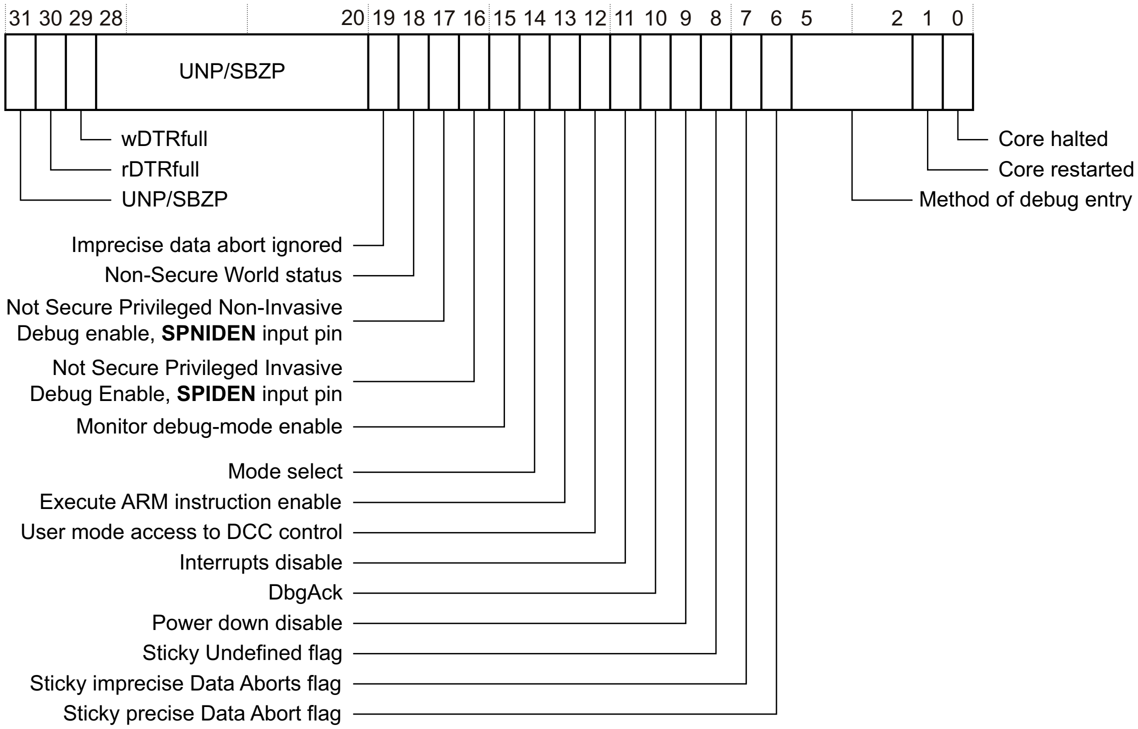
31 30 29 28
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5
2 1 0
UNP/SBZP
wDTRfull
rDTRfull
UNP/SBZP
Imprecise data abort ignored
Non-Secure World status
Not Secure Privileged Non-Invasive Debug enable, **SPNIDEN** input pin
Not Secure Privileged Invasive Debug Enable, **SPIDEN** input pin
Monitor debug-mode enable
Mode select
Execute ARM instruction enable
User mode access to DCC control
Interrupts disable
DbgAck
Power down disable
Sticky Undefined flag
Sticky imprecise Data Aborts flag
Sticky precise Data Abort flag
Core halted
Core restarted
Method of debug entry

Figure 13-3 Debug Status and Control Register format

Table 13-4 lists the bit field definitions for the Debug Status and Control Register.

Table 13-4 Debug Status and Control Register bit field definitions

Bits	Core view	External view	Reset value	Description
[31]	UNP/SBZP	UNP/SBZP	-	Reserved.
[30]	R	R	0	The rDTRfull flag: 0 = rDTR empty 1 = rDTR full. This flag is automatically set on writes by the DBGTAP debugger to the rDTR and is cleared on reads by the core of the same register. No writes to the rDTR are enabled if the rDTRfull flag is set.
[29]	R	R	0	The wDTRfull flag: 0 = wDTR empty 1 = wDTR full. This flag is automatically cleared on reads by the DBGTAP debugger of the wDTR and is set on writes by the core to the same register.
[28:20]	UNP/SBZP	UNP/SBZP	-	Reserved.
[19]	R	R	0	Imprecise Data Aborts Ignored. This read-only bit is set by the core in Debug state following a Data Memory Barrier operation, and cleared on exit from Debug state. When set, the core does not act on imprecise data aborts. However, the sticky imprecise data abort bit is set if an imprecise data abort occurs when in Debug state.

Table 13-4 Debug Status and Control Register bit field definitions (continued)

Bits	Core view	External view	Reset value	Description
[18]	R	R	0	Non-secure World Status bit 0 = The processor is in Secure state. NS bit = 0 or Secure Monitor mode.1 = The processor is in Non-secure state. NS bit = 1 and not Secure Monitor mode.
[17]	R	R	n/a	Not Secure Privilege Non-Invasive Debug Enable, **SPNIDEN**, input pin.0 = SPNIDEN input pin is HIGH.1 = SPNIDEN input pin is LOW.
[16]	R	R	n/a	Not Secure Privilege Invasive Debug Enable, **SPIDEN**, input pin.0 = SPIDEN input pin is HIGH.1 = SPIDEN input pin is LOW.
[15]	RW	R	0	The Monitor debug-mode enable bit: 0 = Monitor debug-mode disabled 1 = Monitor debug-mode enabled. For the core to take a debug exception, Monitor debug-mode has to be both selected and enabled, bit 14 clear and bit 15 set.
[14]	R	RW	0	Mode select bit: 0 = Monitor debug-mode selected 1 = Halting debug-mode selected and enabled.
[13]	R	RW	0	Execute ARM instruction enable bit: 0 = Disabled 1 = Enabled. If this bit is set, the core can be forced to execute ARM instructions in Debug state using the Debug Test Access Port. If this bit is set when the core is not in Debug state, the behavior of the processor is architecturally Unpredictable. For ARM1176JZF-S processors it has no effect.
[12]	RW	R	0	User mode access to comms channel control bit: 0 = User mode access to comms channel enabled 1 = User mode access to comms channel disabled. If this bit is set and a User mode process tries to access the DIDR, DSCR, or the DTR, the Undefined instruction exception is taken. Because accessing the rest of CP14 debug registers is never possible in User mode, see *Executing CP14 debug instructions* on page 13-27, setting this bit means that a User mode process cannot access any CP14 debug register.
[11]	R	RW	0	Interrupts bit: 0 = Interrupts enabled 1 = Interrupts disabled. If this bit is set, the IRQ and FIQ input signals are inhibited.[a]
[10]	R	RW	0	DbgAck bit. If this bit is set, the **DBGACK** output signal (see *External signals* on page 13-52) is forced HIGH, regardless of the processor state.[a]
[9]	R	RW	0	Powerdown disable: 0 = **DBGNOPWRDWN** is LOW 1 = **DBGNOPWRDWN** is HIGH. See *External signals* on page 13-52.

Table 13-4 Debug Status and Control Register bit field definitions (continued)

Bits	Core view	External view	Reset value	Description
[8]	R	RC	0	Sticky Undefined flag: 0 = No Undefined exception trap occurred in Debug state since the last time this bit was cleared. 1 = An undefined exception occurred while in Debug state since the last time this bit was cleared. This bit is cleared on reads of a DBGTAP debugger to the DSCR. The Sticky Undefined bit does not prevent additional instructions from being issued. The Sticky Undefined bit is not set by Undefined exceptions occurring when not in Debug state.
[7]	R	RC	0	Sticky imprecise Data Aborts flag: 0 = No imprecise Data Aborts occurred since the last time this bit was cleared 1 = An imprecise Data Abort has occurred since the last time this bit was cleared. It is cleared on reads of a DBGTAP debugger to the DSCR. The sticky imprecise data abort bit is only set by imprecise data aborts occurring when in Debug state. **Note** In previous versions of the debug architecture, the sticky imprecise data abort was set when the processor took an imprecise data abort. In version 6.1, it is set when an imprecise data abort is detected.
[6]	R	RC	0	Sticky precise Data Abort flag: 0 = No precise Data Abort occurred since the last time this bit was cleared 1 = A precise Data Abort has occurred since the last time this bit was cleared. This flag is meant to detect Data Aborts generated by instructions issued to the processor using the Debug Test Access Port. Therefore, if the DSCR[13] execute ARM instruction enable bit is a 0, the value of the sticky precise Data Abort bit is architecturally Unpredictable. For ARM1176JZF-S processors the sticky precise Data Abort bit is set regardless of DSCR[13]. It is cleared on reads of a DBGTAP debugger to the DSCR. The sticky precise data abort bit is only set by precise data aborts occurring when in Debug state.

Table 13-4 Debug Status and Control Register bit field definitions (continued)

check this on fault
to see the source.

Bits	Core view	External view	Reset value	Description

[5:2]	RW	R	b0000	Method of debug entry bits:
				b0000 = a Halt DBGTAP instruction occurred
				b0001 = a breakpoint occurred
				b0010 = a watchpoint occurred
				b0011 = a BKPT instruction occurred
				b0100 = an **EDBGRQ** signal activation occurred
				b0101 = a vector catch occurred
				b0110 = reserved
				b0111 = reserved
				b1xxx = reserved.
				These bits are set to indicate any of:
				• the cause of a Debug Exception
				• the cause for entering Debug state
				A Prefetch Abort or Data Abort handler must first check the IFSR or DFSR register to determine a debug exception has occurred before checking the DSCR to find the cause. These bits are not set on any events in Debug state.

[1]	R	R	1	Core restarted bit: 0 = the processor is exiting Debug state 1 = the processor has exited Debug state. The DBGTAP debugger can poll this bit to determine when the processor has exited Debug state. See *Debug state* on page 13-37 for a definition of Debug state.
[0]	R	R	0	Core halted bit: 0 = the processor is in normal state 1 = the processor is in Debug state. The DBGTAP debugger can poll this bit to determine when the processor has entered Debug state. See *Debug state* on page 13-37 for a definition of Debug state.

a. Bits DSCR[11:10] can be controlled by a DBGTAP debugger to execute code in normal state as part of the debugging process. For example, if the DBGTAP debugger has to execute an OS service to bring a page from disk into memory, and then return to the application to see the effect this change of state produces, it is undesirable that interrupts are serviced during execution of this routine.

Bits [5:2] are set to indicate:

- the reason for jumping to the Prefetch or Data Abort vector
- the reason for entering Debug state.

A prefetch abort or data abort handler determines if it must jump to the debug monitor target by examining the IFSR or DFSR respectively. A DBGTAP debugger or debug monitor target can determine the specific debug event that caused the Debug state or debug exception entry by examining DSCR[5:2].

13.3.4 CP14 c5, Data Transfer Registers (DTR)

we don't use

This register consists of two separate physical registers:

- the rDTR, Read Data Transfer Register
- the wDTR, Write Data Transfer Register.

The register accessed is dependent on the instruction used:

- writes, MCR and LDC instructions, access the wDTR
- reads, MRC and STC instructions, access the rDTR.

Note

Read and write refer to the core view.

For details of the use of these registers with the rDTRfull flag and wDTRfull flag see *Debug communications channel* on page 13-42. Figure 13-4 shows the format of both the rDTR and wDTR.

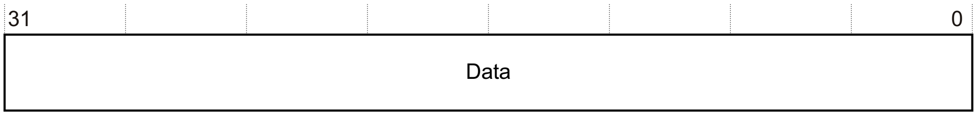

Figure 13-4 DTR format

Table 13-5 lists the bit field definitions for rDTR and wDTR.

Table 13-5 Data Transfer Register bit field definitions

Bits	Core view	External view	Reset value	Description
[31:0]	R	W	-	Read data transfer register, read-only
[31:0]	W	R	-	Write data transfer register, write-only

13.3.5 CP14 c6, Watchpoint Fault Address Register (WFAR)

The purpose of the *Watchpoint Fault Address Register* (WFAR) is to hold the Virtual Address of the instruction that caused the watchpoint.

The register WFAR is:

- in CP14 c6
- a 32-bit read/write register
- accessible in privileged modes only.

When a watchpoint occurs in:

- ARM state, the WFAR contains the address of the instruction causing it plus `0x8`.
- Thumb state, the WFAR contains the address of the instruction causing it plus `0x4`.
- Jazelle state, the WFAR contains the address of the instruction causing it.

The contents of the WFAR are unaffected when a precise Data Abort or Prefetch Abort occurs.

To use the Watchpoint Fault Address Register read or write CP14 with:

- Opcode_1 set to 0
- CRn set to c0
- CRm set to c6
- Opcode_2 set to 0.

For example:

```
MRC p14, 0, <Rd>, c0, c6, 0     ; Read Watchpoint Fault Address Register
MCR p14, 0, <Rd>, c0, c6, 0     ; Write Watchpoint Fault Address Register
```

A write to this register sets the WFAR to the value of the data written. This is useful for a debugger to restore the value of the WFAR.

13.3.6 CP14 c7, Vector Catch Register (VCR)

cute but we do not use. perhaps when building a VMM?

The processor supports efficient exception vector catching. This is controlled by the VCR, as Figure 13-5 shows.

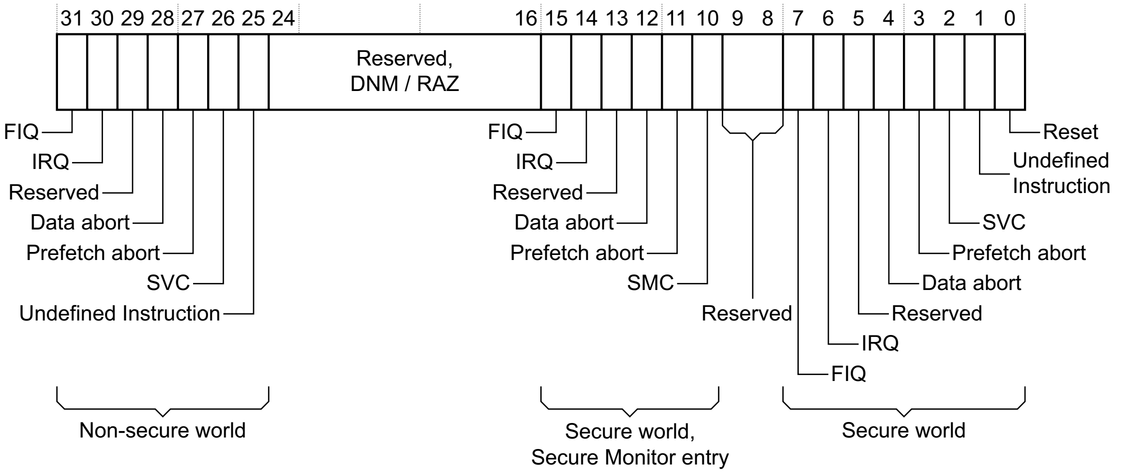

Figure 13-5 Vector Catch Register format

If one of the bits in this register is set and the corresponding vector is committed for execution, then a Debug exception or Debug state entry might be generated, depending on the value of the DSCR[15:14] bits. See *Behavior of the processor on debug events* on page 13-33. Under this model, any kind of fetch of an exception vector can trigger a vector catch, not only the ones because of exception entries.

Vector catches related to bits[15:0] are only triggered by fetches in a Secure world. Catches related to bits [31:25] are only triggered in the Non-secure world.

There are three groups of bits one each to catch exceptions relative to the three vector base address registers for Non-secure, Secure and Secure Monitor modes.

The update of the VCR might occur several instruction after the corresponding MCR instruction. It only takes effect by the next *Instruction Memory Barrier* (IMB).

Bits 29, [24:16], 13, [9:8] and bit 5 are reserved.

Table 13-6 on page 13-14 lists the bit field definitions for the Vector Catch Register. In Table 13-6 on page 13-14, SBA means Secure Base Address, NSBA means Non-secure Base Address, MBA means Monitor Base Address.

Table 13-7 lists the conditions for generation of a Debug exception or entry into Debug State. In this table, SBA means Secure Base Address, NSBA means Non-Secure Base Address, MBA means Monitor Base Address.

Table 13-6 Vector Catch Register bit field definitions

Bits	Read/Write Attributes	Reset value	Vector base	Description
[31]	RW	0	NSBA	Vector Catch Enable - FIQ in Non-secure world.
[30]	RW	0	NSBA	Vector Catch Enable - IRQ in Non-secure world.
[29]	DNM/RAZ	0	-	Reserved
[28]	RW	0	NSBA	Vector Catch Enable - Data Abort in Non-secure world.
[27]	RW	0	NSBA	Vector Catch Enable - Prefetch Abort in Non-secure world.
[26]	RW	0	NSBA	Vector Catch Enable - SVC in Non-secure world.
[25]	RW	0	NSBA	Vector Catch Enable - Undefined Instruction in Non-secure world.
[24:16]	DNM/RAZ	0	-	Reserved
[15]	RW	0	MBA	Vector Catch Enable - FIQ in Secure world.
[14]	RW	0	MBA	Vector Catch Enable - IRQ in Secure world.
[13]	DNM/RAZ	0	-	Reserved
[12]	RW	0	MBA	Vector Catch Enable - Data Abort in Secure world.
[11]	RW	0	MBA	Vector Catch Enable - Prefetch Abort in Secure World
[10]	RW	0	MBA	Vector Catch Enable - SMC in Secure world.
[9:8]	DNM/RAZ	0	-	Reserved
[7]	RW	0	SBA	Vector Catch Enable - FIQ in Secure world.
[6]	RW	0	SBA	Vector Catch Enable - IRQ in Secure world.
[5]	DNM/RAZ	0	-	Reserved
[4]	RW	0	SBA	Vector Catch Enable - Data Abort in Secure world.
[3]	RW	0	SBA	Vector Catch Enable - Prefetch Abort in Secure world.
[2]	RW	0	SBA	Vector Catch Enable, SVC in Secure world.
[1]	RW	0	SBA	Vector Catch Enable, Undefined Instruction in Secure world.
[0]	RW	0	SBA	Vector Catch Enable, Reset

Table 13-7 Summary of debug entry and exception conditions

VCR bit	NS bit, mode	VE	HIVECS	Prefetch vector
VCR[0] = 1	NS bit = 0 or Mode = Secure Monitor.	X	0	`0x00000000`
			1	`0xFFFF0000`

Table 13-7 Summary of debug entry and exception conditions (continued)

VCR bit	NS bit, mode	VE	HIVECS	Prefetch vector
VCR[1] = 1	NS bit = 0 or Mode = Secure Monitor.	X	0	SBA + `0x00000004`
			1	`0xFFFF0004`
VCR[2] = 1	NS bit = 0 or Mode = Secure Monitor.	X	0	SBA + `0x00000008`
			1	`0xFFFF0008`
VCR[3] = 1	NS bit = 0 or Mode = Secure Monitor.	X	0	SBA + `0x0000000C`
			1	`0xFFFF000C`
VCR[4] = 1	NS bit = 0 or Mode = Secure Monitor.	X	0	SBA + `0x00000010`
			1	`0xFFFF0010`
VCR[6] = 1	NS bit = 0 or Mode = Secure Monitor.	0	0	SBA + `0x00000018`
			1	`0xFFFF0018`
		1	X	Most recent Secure IRQ address
VCR[7] = 1	NS bit = 0 or Mode = Secure Monitor.	X	0	SBA + `0x0000001C`
			1	`0xFFFF001C`
VCR[10] = 1	NS bit = 0 or Mode = Secure Monitor.	X	X	MBA + `0x00000008`
VCR[11] = 1	NS bit = 0 or Mode = Secure Monitor.	X	X	MBA + `0x0000000C`
VCR[12] = 1	NS bit = 0 or Mode = Secure Monitor.	X	X	MBA + `0x00000010`
VCR[14] = 1	NS bit = 0 or Mode = Secure Monitor.	X	X	MBA + `0x00000018`
VCR[15] = 1	NS bit = 0 or Mode = Secure Monitor.	X	X	MBA + `0x0000001C`
VCR[25] = 1	NS bit = 1 and mode ≠ Secure Monitor	X	0	NSBA + `0x00000004`
			1	`0xFFFF0004`
VCR[26] = 1	NS bit = 1 and mode ≠ Secure Monitor	X	0	NSBA + `0x00000008`
			1	`0xFFFF0008`
VCR[27] = 1	NS bit = 1 and mode ≠ Secure Monitor	X	0	NSBA + `0x0000000C`
			1	`0xFFFF000C`
VCR[28] = 1	NS bit = 1 and mode ≠ Secure Monitor	X	0	NSBA + `0x00000010`
			1	`0xFFFF0010`

Table 13-7 Summary of debug entry and exception conditions (continued)

VCR bit	NS bit, mode	VE	HIVECS	Prefetch vector
VCR[30] = 1	NS bit = 1 and mode ≠ Secure Monitor	0	0	NSBA + `0x00000018`
			1	`0xFFFF0018`
		1	X	Most recent Non-secure IRQ address.
VCR[31] = 1	NS bit = 1 and mode ≠ Secure Monitor	X	0	NSBA + `0x0000001C`
			1	`0xFFFF001C`

13.3.7 CP14 c64-c69, Breakpoint Value Registers (BVR)

Table 13-8 lists the Breakpoint Value Registers that the processor implements.

Table 13-8 Processor breakpoint and watchpoint registers

Binary address		Register number	CP14 debug register name	Abbreviation	Context ID capable?
Opcode_2	CRm				
b100	b0000-b0011	c64-c67	Breakpoint Value Registers 0-3	BVR0-3	No
	b0100-b0101	c68-c69	Breakpoint Value Registers 4-5	BVR4-5	Yes

confusing notation: you encode c64 as c0, c65 as c1, etc.

Each BVR is associated with a BCR register. BCRy is the corresponding control register for BVRy.

A pair of breakpoint registers, BVRy/BCRy, is called a *Breakpoint Register Pair* (BRP). BVR0-5 are paired with BCR0-5 to make BRP0-5.

The BVR of a BRP is loaded with an IMVA and then its contents can be compared against the IMVA bus of the processor. The breakpoint value contained in the BVR corresponds to either an IMVA or a context ID. Breakpoints can be set on:

- an IMVA
- a context ID
- an IMVA/context ID pair.

The IMVA comparison can be programmed to either hit when the address matches or mis-matches. The IMVA mis-match case is useful because it enables a debugger to implement a single-step operation when the breakpoint is programmed to match any other IMVA than the instruction about to be executed.

fantastic since you do not have to understand instruction semantics. HOWEVER:

The processor supports thread-aware breakpoints and watchpoints. A context ID can be loaded into the BVR and the BCR can be configured so this BVR value is compared against the CP15 Context ID Register, c13, instead of the IMVA bus. Another register pair loaded with an IMVA or DMVA can then be linked with the context ID holding BRP. A breakpoint or watchpoint debug event is only generated if both the address and the context ID match at the same time. This means that unnecessary hits can be avoided when debugging a specific thread within a task.

typical architecture manual: buries key sentence about when an ability does not work. i lost an hour trying to figure out what was going on before re-reading to see if i missed something along these lines.

Breakpoint debug events generated on context ID matches only are also supported. However, if a context ID only match or any match including an IMVA mis-match occurs while the processor is running in a privileged mode and the debug logic in Monitor debug-mode, it is ignored. This is to avoid the processor ending in an unrecoverable state.

Table 13-9 lists the bit field definitions for context ID and non context ID Breakpoint Value Registers.

Table 13-9 Breakpoint Value Registers, bit field definition

Context ID capable?	Bits	Read/write attributes	Description
No	[31:2]	RW	Breakpoint address
Yes	[31:0]	RW	Breakpoint address or context ID

When a context ID capable BRP is set for IMVA comparison, BVR bits [1:0] are ignored.

13.3.8 CP14 c80-c85, Breakpoint Control Registers (BCR)

These registers contain the necessary control bits for setting:

- breakpoints
- linked breakpoints.

Table 13-10 lists the Breakpoint Control Registers and that the processor implements.

Table 13-10 Processor Breakpoint Control Registers

Binary address		Register number	CP14 debug register name	Abbreviation	Context ID capable?
Opcode_2	CRm				
b101	b0000-b0011	c80-c83	Breakpoint Control Registers 0-3	BCR0-3	No
	b0100-b0101	c84-c85	Breakpoint Control Registers 4-5	BCR4-5	Yes

Figure 13-6 shows the format of the Breakpoint Control Registers.

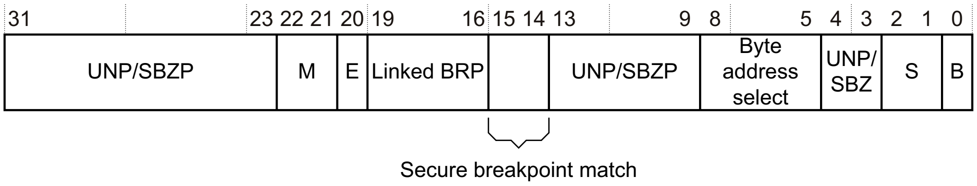

Figure 13-6 Breakpoint Control Registers, format

Table 13-11 lists the bit field definitions for the Breakpoint Control Registers.

Table 13-11 Breakpoint Control Registers, bit field definitions

Bits	Read/write attributes	Reset value	Description
[31:23]	UNP/SBZP	-	Reserved.
[22:21]	RW	00	Meaning of BVR00 = IMVA Match.01 = Context ID Match.10 = IMVA Mis-match.11 = Reserved. If this breakpoint does not have Context ID capability, bit 21 is RAZ.
[20]	RW	-	Enable linking: 0 = Linking disabled 1 = Linking enabled. When this bit is set HIGH, the corresponding BRP is linked. See Table 13-12 on page 13-19 for details.
[19:16]	RW	-	Linked BRP number. The binary number encoded here indicates another BRP to link this one with. If a BRP is linked with itself, it is architecturally Unpredictable if a breakpoint debug event is generated. For ARM1176JZF-S processors the breakpoint debug event is not generated.
[15:14]	RW	-	b00 = Breakpoint matches in Secure or Non-secure world. b01 = Breakpoint only matches in Non-secure world. b10 = Breakpoint only matches in Secure world.b11 = Reserved If this BRP is programmed for context ID comparison and linking (BCR[22:20] is set b011), then the BCR[15:14] field of the IMVA-holding BRP takes precedence and it is Undefined whether this field is included in the comparison or not. Therefore, it must be set to b00. The WCR[15:14] field of a WRP linked with this BRP also takes precedence over this field.
[13:9]	UNP/SBZP	-	Reserved.
[8:5]	RW	-	Byte address select. The BVR is programmed with a word address. You can use this field to program the breakpoint so it matches only if certain byte addresses are accessed. b0000 = The breakpoint never matches bxxx1= If the byte at address {BVR[31:2], b00}+0 is accessed, the breakpoint matches bxx1x = If the byte at address {BVR[31:2], b00}+1 is accessed, the breakpoint matches bx1xx = If the byte at address {BVR[31:2], b00}+2 is accessed, the breakpoint matches b1xxx = If the byte at address {BVR[31:2], b00}+3 is accessed, the breakpoint matches. This field must be set to b1111 when this BRP is programmed for context ID comparison, that is BCR[22:20] set to b01x. Otherwise breakpoint or watchpoint debug events might not be generated as expected. **Note** These are little-endian byte addresses. This ensures that a breakpoint is triggered regardless of the endianness of the instruction fetch. For example, if a breakpoint is set on a certain Thumb instruction by doing BCR[8:5] = b0011, it is triggered if in little-endian and IMVA[1:0] is b00 or if big-endian and IMVA[1:0] is b10.

we don't use linking

key: if you don't set this up, won't match even if the breakpoint is enabled.

Table 13-11 Breakpoint Control Registers, bit field definitions (continued)

Bits	Read/write attributes	Reset value	Description
[4:3]	UNP/SBZP	-	Reserved

[2:1]	RW	-	Supervisor Access. The breakpoint can be conditioned to the privilege of the access being done:
			b00 = Reserved
			b01= Privileged
			b10 = User
			b11 = Either.
			If this BRP is programmed for context ID comparison and linking, BCR[22:20] is set b011, then the BCR[2:1] field of the IMVA-holding BRP takes precedence and it is Undefined whether this field is included in the comparison or not. Therefore, it must be set to either.
			The WCR[2:1] field of a WRP linked with this BRP also takes precedence over this field.

[0]	RW	0	Breakpoint enable: 0 = Breakpoint disabled 1 = Breakpoint enabled.

Table 13-12 summarizes the meaning of BCR bits [22:20].

Table 13-12 Meaning of BCR[22:20] bits

BCR[22:20]	Meaning
b000	The corresponding BVR is compared against the IMVA bus. This BRP is not linked with any other one. It generates a breakpoint debug event on an IMVA match.
b001	The corresponding BVR is compared against the IMVA bus. This BRP is linked with the one indicated by BCR[19:16] linked BRP field. They generate a breakpoint debug event on a joint IMVA and context ID match.
b010	The corresponding BVR is compared against CP15 Context Id Register, c13. This BRP is not linked with any other one. It generates a breakpoint debug event on a context ID match.
b011	The corresponding BVR is compared against CP15 Context Id Register, c13. Another BRP, of the BCR[21:20]=b01 type, or WRP, with WCR[20]=b1, is linked with this BRP. They generate a breakpoint or watchpoint debug event on a joint IMVA or DMVA and context ID match.
b100	The corresponding BVR is compared against the IMVA bus. This BRP is not linked with any other one. It generates a breakpoint debug event on an IMVA mismatch.
b101	The corresponding BVR is compared against the IMVA bus. This BRP is linked with the one indicated by BCR[19:16] linked BRP field. They generate a breakpoint debug event on a joint IMVA mismatch and context ID match.
b110	Reserved
b111	Reserved

i initially thought we couldn't do w/o virtual memory since (usually!) you can't run at user-level without. however, very weirdly, it appears on the arm you don't need VM to run at user. will investigate further.

Note

- The BCR[8:5], BCR[15:14], and BCR[2:1] fields still apply when a BRP is set for context ID comparison. See *Setting breakpoints, watchpoints, and vector catch debug events* on page 13-45 for detailed programming sequences for linked breakpoints and linked watchpoints.

- The BCR[8:5] field is treated as part of the compared address, For an IMVA mismatch the bits must be set to 1 for the corresponding byte lanes that are excluded from the breakpoint.

The following rules apply to the processor for breakpoint debug event generation:

must do this. easy to miss. "but it worked"

- The update of a BVR or a BCR can take effect several instructions after the corresponding MCR. It takes effect by the next IMB.
- Updates of the CP15 Context ID Register c13, can take effect several instructions after the corresponding MCR. However, the write takes place by the end of the exception return. This is to ensure that a User mode process, switched in by a processor scheduler, can break at its first instruction.
- Any BRP, holding an IMVA, can be linked with any other one with context ID capability. Several BRPs, holding IMVAs, can be linked with the same context ID capable one.
- If a BRP, holding an IMVA, is linked with one that is not configured for context ID comparison and linking, it is architecturally Unpredictable whether a breakpoint debug event is generated or not. For ARM1176JZF-S processors the breakpoint debug event is not generated. BCR[22:20] fields of the second BRP must be set to b011.
- If a BRP, holding an IMVA, is linked with one that is not implemented, it is architecturally Unpredictable if a breakpoint debug event is generated or not. For ARM1176JZF-S processors the breakpoint debug event is not generated.
- If a BRP is linked with itself, it is architecturally Unpredictable if a breakpoint debug event is generated or not. For ARM1176JZF-S processors the breakpoint debug event is not generated.
- If a BRP, holding an IMVA, is linked with another BRP, holding a context ID value, and they are not both enabled, both BCR[0] bits set, the first one does not generate any breakpoint debug event.

13.3.9 CP14 c96-c97, Watchpoint Value Registers (WVR)

Each WVR is associated with a WCR register. WCRy is the corresponding register for WVRy.

A pair of watchpoint registers, WVRy and WCRy, is called a *Watchpoint Register Pair* (WRP). WVR0-1 are paired with WCR0-1 to make WRP0-1.

Table 13-13 lists the Watchpoint Value Registers that the processor implements.

Table 13-13 Processor Watchpoint Value Registers

Binary address		Register number	CP14 debug register name	Abbreviation	Context ID capable?
Opcode_2	CRm				
b110	b0000-b0001	c96-c97	Watchpoint Value Registers 0-1	WVR0-1	-

The watchpoint value contained in the WVR always corresponds to a DMVA. Watchpoints can be set on:

- a DMVA
- a DMVA/context ID pair.

For the second case a WRP and a BRP with context ID comparison capability have to be linked. A debug event is generated when both the DMVA and the context ID pair match simultaneously. Table 13-14 lists the bit field definitions for the Watchpoint Value Registers.

Table 13-14 Watchpoint Value Registers, bit field definitions

Bits	Read/write attributes	Reset value	Description
[31:2]	RW	-	Watchpoint address
[1:0]	UNP/SBZP	-	-

you should check this in the given address.

13.3.10 CP14 c112-c113, Watchpoint Control Registers (WCR)

These registers contain the necessary control bits for setting:

- watchpoints
- linked watchpoints.

Table 13-15 lists the Watchpoint Control Registers that the processor implements.

Table 13-15 Processor Watchpoint Control Registers

Binary address		Register number	CP14 debug register name	Abbreviation	Context ID capable?
Opcode_2	CRm				
b111	b0000-b0001	c112-c113	Watchpoint Control Registers 0-1	WCR0-1	-

Figure 13-7 shows the format of the Watchpoint Control Registers.

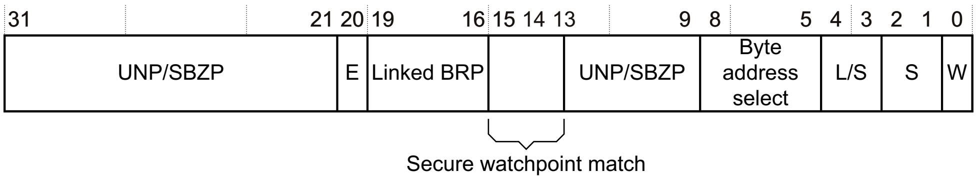

Figure 13-7 Watchpoint Control Registers, format

Table 13-16 lists the bit field definitions for the Watchpoint Control Registers.

Table 13-16 Watchpoint Control Registers, bit field definitions

Bits	Read/write attributes	Reset value	Description
[31:21]	UNP/SBZP	-	Reserved.
[20]	RW	-	Enable linking bit: 0 = Linking disabled 1 = Linking enabled. When this bit is set, this watchpoint is linked with the context ID holding BRP selected by the linked BRP field.
[19:16]	RW	-	Linked BRP. The binary number encoded here indicates a context ID holding BRP to link this WRP with.

Table 13-16 Watchpoint Control Registers, bit field definitions (continued)

Bits	Read/write attributes	Reset value	Description

[15:14]	RW	-	b00 = Watchpoint matches in Secure or Non-secure world.
			b01 = Watchpoint only matches in Non-secure world.
			b10 = Watchpoint only matches in Secure world.
			b11 = Reserved.

[13:9]	SBZ	-	Reserved.
[8:5]	RW	-	Byte address select. The WVR is programmed with a word address. This field can be used to program the watchpoint so it hits only if certain byte addresses are accessed.b0000 = The watchpoint never hits bxxx1= If the byte at address {WVR[31:2], b00}+0 is accessed, the watchpoint hitsbxx1x = If the byte at address {WVR[31:2], b00}+1 is accessed, the watchpoint hitsbx1xx = If the byte at address {WVR[31:2], b00}+2 is accessed, the watchpoint hitsb1xxx = If the byte at address {WVR[31:2], b00}+3 is accessed, the watchpoint hits. **Note** These are little-endian byte addresses. This ensures that a watchpoint is triggered regardless of the way it is accessed. For example, if a watchpoint is set on a certain byte in memory by doing WCR[8:5] = b0001. `LDRB R0, #0x0` it triggers the watchpoint in little-endian mode, as does `LDRB R0, #x3` in legacy big-endian mode, B bit of CP15 c1 set.

[4:3]	RW	-	Load/store access. The watchpoint can be conditioned to the type of access being done:
			b00 = Reserved
			b01 = Load
			b10 = Store
			b11 = Either.
			A SWP triggers on Load, Store, or Either. Load exclusive instructions, LDREX, LDREXB, LDREXD, and LDREXH, trigger on Load or Either. Store exclusive instructions, STREX, STREXB, STREXD, and STREXH, trigger on Store or Either, whether it succeeded or not.

[2:1]	RW	-	Supervisor Access. The watchpoint can be conditioned to the privilege of the access being done:
			b00 = Reserved
			b01 = Privileged
			b10 = User
			b11 = Either.

[0]	RW	0	Watchpoint enable: 0 = Watchpoint disabled 1 = Watchpoint enabled.

need to set or won't match
no matter the rest of the
config.

very cool.

In addition to the rules for breakpoint debug event generation, see *CP14 c80-c85, Breakpoint Control Registers (BCR)* on page 13-17, the following rules apply to the processor for watchpoint debug event generation:

- The update of a WVR or a WCR can take effect several instructions after the corresponding MCR. It only guaranteed to have taken effect by the next IMB.

must do an IMB!

- Any WRP can be linked with any BRP with context ID comparison capability. Several BRPs, holding IMVAs, and WRPs can be linked with the same context ID capable BRP.
- If a WRP is linked with a BRP that is not configured for context ID comparison and linking, it is architecturally Unpredictable if a watchpoint debug event is generated or not. For ARM1176JZF-S processors the watchpoint debug event is not generated. BCR[22:20] fields of the BRP must be set to b011.
- If a WRP is linked with a BRP that is not implemented, it is architecturally Unpredictable if a watchpoint debug event is generated or not. For ARM1176JZF-S processors the watchpoint debug event is not generated.
- If a WRP is linked with a BRP and they are not both enabled, BCR[0] and WCR[0] set, it does not generate a watchpoint debug event.

13.3.11 CP14 c10, Debug State Cache Control Register

The Debug State Cache Control Register controls cache behavior in Debug state:

```
MRC p14, 0, <Rd>, c0, c10, 0
MCR p14, 0, <Rd>, c0, c10, 0
```

Table 13-17 lists the functional bits in the register.

Table 13-17 Debug State Cache Control Register bit functions

Bits	Reset value	Name	Description
[31:3]	UNP/SBZ	-	Reserved.
[2]	0	nWT	Not Write-Through: 1 = Normal operation of regions marked as Write-Back in Debug state. 0 = force Write-Through behavior for regions marked as Write-Back in Debug state.
[1]	0	nIL	No Instruction Cache Line-Fill: 1 = Normal operation of Instruction Cache line fills in Debug state. 0 = Instruction Cache line-fill disabled in Debug state.
[0]	0	nDL	No Data/Unified Cache Line-Fill: 1 = Normal operation of Data/Unified Cache line-fills in Debug state. 0 = Data/Unified Cache line-fill disabled in Debug state.

The effect of these bits only applies in Debug state. The operation under control only occurs if it is enabled in both this register and by the corresponding bit in the Cache Behavior Override Register.

13.3.12 CP14 c11, Debug State MMU Control Register

The Debug State MMU Control Register controls main and micro TLB behavior in Debug state:

```
MRC p14, 0, <Rd>, c0, c11, 0
MCR p14, 0, <Rd>, c0, c11, 0
```

Table 13-18 on page 13-24 lists the functional bits in the register.

Table 13-18 Debug State MMU Control Register bit functions

Bits	Reset value	Name	Description
[31:7]	UNP/SBZ	-	Reserved
[6]	0	nDMM	1 = Normal operation of Main TLB matching in Debug state. 0 = Main TLB match disabled in Debug state.
[5]	UNP/SBZ	-	Reserved
[4]	0	nDML	1 = Normal operation of Main TLB loading in Debug state. 0 = Main TLB load disabled in Debug state.
[3]	0	nIUM	1 = Normal operation of Instruction Micro TLB matching in Debug state. 0 = Instruction Micro TLB match disabled in Debug state.
[2]	0	nDUM	1 = Normal operation of Data Micro TLB matching in Debug state. 0 = Data Micro TLB match disabled in Debug state.
[1]	0	nIUL	1 = Normal operation of Instruction Micro TLB loading and flushing in Debug state. 0 = Instruction Micro TLB load and flush disabled in Debug state.
[0]	0	nDUL	1 = Normal operation of Data Micro TLB loading and flushing in Debug state. 0 = Data Micro TLB load and flush disabled in Debug state.

13.4 CP14 registers reset

The CP14 debug registers that are accessible through the external interface are all reset by the processor power-on reset signal, **nPORESETIN**, see *Reset with no IEM* on page 9-4 or *Reset with IEM* on page 9-8.

This ensures that a vector catch set on the reset vector is taken when **nRESETIN** is deasserted. It also ensure that the DBGTAP debugger can be connected when the processor is running without clearing CP14 debug setting, because **DBGnTRST** does not reset these registers.

13.5 CP14 debug instructions

Table 13-19 lists the CP14 debug instructions.

cheat sheet of assembly.

Table 13-19 CP14 debug instructions

Binary address		Register number	Abbreviation	Legal instructions
Opcode_2	CRm			
b000	b0000	0	DIDR	`MRC p14, 0, <Rd>, c0, c0, 0`[a]
b000	b0001	1	DSCR	`MRC p14, 0, <Rd>, c0, c1,0`[a] `MRC p14, 0, R15, c0, c1,0` `MCR p14, 0, <Rd>, c0, c1,0`[a]

b000	b0101	5	DTR (rDTR/wDTR)	`MRC p14, 0, <Rd>, c0, c5, 0`[a]
				`MCR p14, 0, <Rd>, c0, c5, 0`[a]
				`STC p14, c5, <addressing mode>`
				`LDC p14, c5, <addressing mode>`

b000	b0110	6	WFAR	`MRC p14, 0, <Rd>, c0, c6, 0`[a] `MCR p14, 0, <Rd>, c0, c6, 0`[a]
b000	b0111	7	VCR	`MRC p14, 0, <Rd>, c0, c7, 0`[a] `MCR p14, 0, <Rd>, c0, c7, 0`[a]
b000	b1010	10	DSCCR	`MRC p14, 0, <Rd>, c0, c10, 0`[a] `MCR p14, 0, <Rd>, c0, c10, 0`[a]
b000	b1011	11	DSMCR	`MRC p14, 0, <Rd>, c0, c11, 0`[a] `MCR p14, 0, <Rd>, c0, c11, 0`[a]
b100	b0000-b1111	64-79	BVR	`MRC p14, 0, <Rd>, c0, cy,4`[a][b] `MCR p14, 0, <Rd>, c0, cy,4`[a][b]
b101	b0000-b1111	80-95	BCR	`MRC p14, 0, <Rd>, c0, cy,5`[a][b] `MCR p14, 0, <Rd>, c0, cy,5`[a][b]
b110	b0000-b1111	96-111	WVR	`MRC p14, 0, <Rd>, 0, cy, 6`[a][b] `MCR p14, 0, <Rd>, 0, cy, 6`[a][b]
b111	b0000-b1111	112-127	WCR	`MRC p14, 0, <Rd>, c0, cy, 7`[a][b] `MCR p14, 0, <Rd>, c0, cy, 7`[a][b]

[a] <Rd> is any of R0-R14 ARM registers.

[b] y is the decimal representation for the binary number CRm.

In Table 13-19, `MRC p14,0,<Rd>,c0,c5,0` and `STC p14,c5,<addressing mode>` refer to the rDTR and `MCR p14,0,<Rd>,c0,c5,0` and `LDC p14,c5,<addressing mode>` refer to the wDTR. See *CP14 c5, Data Transfer Registers (DTR)* on page 13-11 for more details. The `MRC p14,0,R15,c0,c1,0` instruction sets the CPSR flags as follows:

- N flag = DSCR[31]. This is an Unpredictable value.
- Z flag = DSCR[30]. This is the value of the rDTRfull flag.
- C flag = DSCR[29]. This is the value of the wDTRfull flag.
- V flag = DSCR[28]. This is an Unpredictable value.

Use of R15 in all other MRC instructions that Table 13-19 on page 13-26 lists, sets all four flags to Unpredictable values.

Instructions that follow the MRC instruction can be conditioned to these CPSR flags.

13.5.1 Executing CP14 debug instructions

If the core is in Debug state, see *Debug state* on page 13-37, you can execute any CP14 debug instruction regardless of the processor mode.

If the processor tries to execute a CP14 debug instruction that either is not in Table 13-19 on page 13-26, or is targeted to a reserved register, such as a non-implemented BVR, the Undefined instruction exception is taken.

You can access the DCC, read DIDR, read DSCR and read/write DTR, in User mode. All other CP14 debug instructions are privileged. If the processor tries to execute one of these in User mode, the Undefined instruction exception is taken.

If the User mode access to DCC disable bit, DSCR[12], is set, all CP14 debug instructions are considered as privileged, and all attempted User mode accesses to CP14 debug registers generate an Undefined instruction exception.

When DSCR bit 14 is set, Halting debug-mode selected and enabled, if the software running on the processor tries to access any register other than the DIDR, the DSCR, or the DTR, the core takes the Undefined instruction exception. The same thing happens if the core is not in any Debug mode, DSCR[15:14]=b00. This lockout mechanism ensures that the software running on the core cannot modify the settings of a debug event programmed by the DBGTAP debugger.

Table 13-20 lists the results of executing CP14 debug instructions.

Table 13-20 Debug instruction execution

State when executing CP14 debug instruction:				**Results of CP14 debug instruction execution:**		
Processor mode	**Debug state**	**DSCR[15:14], Mode enabled and selected**	**DSCR[12], DCC User accesses disabled**	**Read DIDR, read DSCR and read/ write DTR**	**Write DSCR**	**Read/write other debug registers**
x	Yes	xx	x	Proceed	Proceed	Proceed
User	No	xx	0	Proceed	Undefined exception	Undefined exception
User	No	xx	1	Undefined exception	Undefined exception	Undefined exception
Privileged	No	b00, None	x	Proceed	Proceed	Undefined exception
Privileged	No	b01, Halting	x	Proceed	Proceed	Undefined exception
Privileged	No	b10, Monitor	x	Proceed	Proceed	Proceed
Privileged	No	b11, Halting	x	Proceed	Proceed	Undefined exception

~~13.6~~ External debug interface

The debug architecture provides two control signals called **SPIDEN** and **SPNIDEN**. that are part of the external debug interface.

SPIDEN The Secure Privileged Invasive Debug Enable input pin, **SPIDEN**, that enables and disables invasive debug in the Secure world:

- If this input signal is HIGH, invasive debug is permitted in all Secure modes. In this case invasive debug is permitted in Secure User mode, regardless value of SUIDEN bit.
- If this input signal is LOW, invasive debug is not permitted in any Secure privileged mode. Invasive debug is permitted in Secure User mode according to the SUIDEN bit.

SPNIDEN The Secure Privileged Non-Invasive Debug Enable input pin, **SPNIDEN**, that enables and disables non-invasive debug in the Secure world:

- If this input signal is HIGH, non-invasive debug is permitted in all Secure modes. In this case non-invasive debug is permitted in Secure User mode, regardless of the value of the SUNIDEN bit.
- If this input signal is LOW, non-invasive debug is not permitted in all Secure privileged modes. Non-invasive debug is permitted in Secure User mode according to the SUNIDEN bit.

Note

- You must control access to the **SPIDEN** and **SPNIDEN** pins, as they represent a significant security risk. For example, it must not be possible to set these pins through the boundary scan in a final device.
- For software systems that do not use any TrustZone security features, the **SPIDEN** and **SPNIDEN** pins must be driven HIGH to enable debug by default.

Table 13-21 lists the relationship between the **DBGEN** input pin, the **SPIDEN** input pin, the SUIDEN control bit, the NS bit, the processor mode and the debug capabilities.

Table 13-21 Secure debug behavior

DBGEN	DSCR [15:14]	SPIDEN	SUIDEN	NS bit	Mode	Debug-mode	Notes
0	XX	X	X	X	X	Debug disabled.	DSCR[15:14] reads as zero
1	00	1	X	X	X	No debug mode selected[a]	Permitted in Non-secure state and in all modes in Secure state.
1	00	0	0	1	not Secure Monitor	No debug mode selected[a]	Permitted only in Non-secure state.
1	00	0	0	X	Secure Monitor	Debug not permitted[b]	Not permitted in Secure state.
1	00	0	0	0	X	Debug not permitted[b]	Not permitted in Secure state.
1	00	0	1	1	not Secure Monitor	No debug mode selected[a]	Permitted in Non-secure state.

Table 13-21 Secure debug behavior (continued)

DBGEN	DSCR [15:14]	SPIDEN	SUIDEN	NS bit	Mode	Debug-mode	Notes
1	00	0	1	X	Secure Monitor	Debug not permitted[b]	Not permitted in privileged modes in Secure state.
1	00	0	1	0	not User	Debug not permitted[b]	Not permitted in privileged modes in Secure state.
1	00	0	1	0	User	No debug mode selected[a]	Permitted in User mode in Secure state.[c]
1	10	1	X	X	X	Monitor debug-mode	Permitted in Non-secure state and in all modes in Secure state.
1	10	0	0	1	not Secure Monitor	Monitor debug-mode	Permitted only in Non-secure state.
1	10	0	0	X	Secure Monitor	Debug not permitted[b]	Not permitted in Secure state.
1	10	0	0	0	X	Debug not permitted[b]	Not permitted in Secure state.
1	10	0	1	1	not Secure Monitor	Monitor debug-mode	Permitted in Non-secure state.
1	10	0	1	X	Secure Monitor	Debug not permitted[b]	Not permitted in privileged modes in Secure state.
1	10	0	1	0	not User	Debug not permitted[b]	Not permitted in privileged modes in Secure state.
1	10	0	1	0	User	Monitor debug-mode	Permitted in User mode in Secure state.[c]
1	X1	1	X	X	X	Halting debug-mode	Permitted in Non-secure state and in all modes in Secure state.
1	X1	0	0	1	not Secure Monitor	Halting debug-mode	Permitted in Non-secure state.
1	X1	0	0	X	Secure Monitor	Debug not permitted[b]	Not permitted in Secure state.
1	X1	0	0	0	X	Debug not permitted[b]	Not permitted in Secure state.
1	X1	0	1	1	not Secure Monitor	Halting debug-mode	Permitted in Non-secure state.
1	X1	0	1	X	Secure Monitor	Debug not permitted[b]	Not permitted in privileged modes in Secure state.
1	X1	0	1	0	not User	Debug not permitted[b]	Not permitted in privileged modes in Secure state.
1	X1	0	1	0	User	Halting debug-mode	Permitted in User mode in Secure state. Capabilities restricted.

a. *Behavior of the processor on debug events* on page 13-33 describes the behavior when no debug mode is selected. Only the BKPT instruction external debug request signal, and Halt DBGTAP instructions have an effect when no debug mode is selected. All other debug events are ignored.

b. *Behavior of the processor on debug events* on page 13-33 describes the behavior marked as not permitted. Logically, the processor is still configured for either Halting debug-mode or Monitor debug-mode, as appropriate.

c. Debug exceptions are handled in a privileged mode.

~~13.7~~ Changing the debug enable signals

The behavior of these control signals, **DBGEN**, **SPIDEN**, and **SPNIDEN**, is primarily a concern of the external debug interface. It is recommended that these signals do not change. However, the architecture permits these signals to change when the processor is running or when the processor is in Debug state.

If software running on the processor changes the state of one of these signals, before performing debug or analysis operations that rely on the new value it must:

1. Execute the device specific sequence of instructions to change the signal value. For instance, the software might have to write a value to a control register in a system peripheral.
2. Perform a Data Memory Barrier operation. This stage can be omitted if the previous stage does not involve any memory operations.
3. Poll debug registers for the view that the processor has of the signal values. This stage is required because system specific issues might result in the processor not receiving a signal change until some cycles after the Data Memory Barrier completes.
4. Issue an Instruction Memory Barrier sequence.

The same rules apply for instructions executed through the ITR when in Debug state.

The view that the processor has of the **SPIDEN** and **SPNIDEN** signals can be polled through the DSCR. The processor has no register that shows its view of **DBGEN**. However, if **DBGEN** is LOW, DSCR[15:14] read as zero, and therefore the view that the processor has of **DBGEN** can be polled by writing to DSCR[15:14] and using the value read back to determine its setting.

13.8 Debug events

A debug event is any of the following:

- *Software debug event*
- *External debug request signal*
- *Halt DBGTAP instruction* on page 13-33.

13.8.1 Software debug event

A software debug event is any of the following:

summary of previous
information

- A watchpoint debug event. This occurs when:
  - — the DMVA present in the data bus matches the watchpoint value
  - — all the conditions of the WCR match
  - — the watchpoint is enabled
  - — the linked contextID-holding BRP, if any, is enabled and its value matches the context ID in CP15 c13.
- A breakpoint debug event. This occurs when:
  - — an instruction was fetched and the IMVA present in the instruction bus matched or mismatched the breakpoint value, according to the meaning field in the BCR
  - — at the same time the instruction was fetched, all the conditions of the BCR matched
  - — the breakpoint was enabled
  - — at the same time the instruction was fetched, the linked contextID-holding BRP, if any, was enabled and its value matched the context ID in CP15 c13
  - — the instruction is now committed for execution.
- A breakpoint debug event also occurs when:
  - — an instruction was fetched and the CP15 Context ID, register 13, matched the breakpoint value
  - — at the same time the instruction was fetched, all the conditions of the BCR matched
  - — the breakpoint was enabled
  - — the instruction is now committed for execution.

we could have
done this vs branches
in the memcheck stat lab.

- A software breakpoint debug event. This occurs when a BKPT instruction is committed for execution.
- A vector catch debug event. This occurs when:
  - — The instruction at a vector location was fetched in the appropriate Secure or Non-secure world. This includes any kind of prefetches, not only the ones because of exception entry.
  - — At the same time the instruction was fetched, the corresponding bit of the VCR was set, vector catch enabled.
  - — The instruction is now committed for execution.

13.~~8~~.2 External debug request signal

The processor has an external debug request input signal, **EDBGRQ**. When this signal is HIGH it causes the processor to enter Debug state when execution of the current instruction has completed. When this happens, the DSCR[5:2] method of entry bits are set to b0100.This signal can be driven by the ETM to signal a trigger to the core. For example, if a memory permission fault occurs, an external Trace analyzer can collect trace information around this trigger event at the same time that the processor is stopped to examine its state. See the *Chapter 15 Trace Interface Port* for more details. A DBGTAP debugger can also drive this signal.

13.8.3 Halt DBGTAP instruction

The Halt mechanism is used by the Debug Test Access Port to force the core into Debug state. When this happens, the DSCR[5:2] method of entry bits are set to b0000.

13.8.4 Behavior of the processor on debug events

This section describes how the processor behaves on debug events while not in Debug state. See *Debug state* on page 13-37 for information on how the processor behaves while in Debug state. When a software debug event occurs and Monitor debug-mode is selected and enabled and the core is in a state that permits debug then a Debug exception is taken. However, Prefetch Abort and Data Abort Vector catch debug events are ignored.

This is to avoid the processor ending in an unrecoverable state on certain combinations of exceptions and vector catches. Unlinked context ID and all address mismatch breakpoint debug events are also ignored if the processor is running in a privileged mode and Monitor debug-mode is selected and enabled.

When the external debug request signal is activated, or the DBGTAP instruction is issued and debug is enabled by **DBGEN** and the core is in a state that permits debug, the processor enters Debug state regardless of any debug-mode selected by DSCR[15:14].

When a debug event occurs and Halting debug-mode is selected and enabled and the core is in a state that debug is permitted, then the processor enters Debug state.

All software debug events other than the BKPT instruction, that is register breakpoints, watchpoints, and vector catches, when no debug mode is selected and enabled or the core is in a state that does not permit debug, are ignored.

When neither Halting nor Monitor debug-mode is selected and enabled or the core is in a state that does not permit debug, the BKPT instruction generates a Prefetch Abort exception. Table 13-22 lists the behavior of the processor in debug events.

Table 13-22 Behavior of the processor on debug events

DBGEN	DSCR[15:14]	Mode selected, enabled and permitted	Action on software debug event	Action on external debug request signal activation	Action on Halt DBGTAP
0	bxx	-[a]	Ignore/Prefetch Abort[b]	Ignore	Ignore
1	b00	None	Ignore/Prefetch Abort[a]	Debug state entry	Debug state entry
1	b01	Halting	Debug state entry	Debug state entry	Debug state entry
1	b10	Monitor	Debug exception/Ignore[c]	Debug state entry	Debug state entry
1	b11	Halting	Debug state entry	Debug state entry	Debug state entry

[a] Entry to Debug state is disabled.

[b] When no debug mode is selected and enabled or the core is in a state that does not permit debug, a BKPT instruction generates a Prefetch Abort exception instead of being ignored.

[c] Prefetch Abort and Data Abort vector catch debug events are ignored in Monitor debug-mode. Unlinked context ID and address mismatch breakpoint debug events are also ignored if the processor is running in a privileged mode and Monitor debug-mode is selected and enabled.

13.8.5 Effect of a debug event on CP15 registers

The four CP15 registers that can be set on a debug event are:

- *Instruction Fault Status Register* (IFSR)
- *Data Fault Status Register* (DFSR)
- *Fault Address Register* (FAR)
- *Watchpoint Fault Address Register* (WFAR).

The *Instruction Fault Address Register* (IFAR) is never updated on debug events.

The registers are set under the following circumstances:

- The IFSR is set whenever a breakpoint, software breakpoint, or vector catch debug event generates a Debug exception entry. It is set to indicate the cause for the Prefetch Abort vector fetch.
- The DFSR is set whenever a watchpoint debug event generates a Debug exception entry. It is set to indicate the cause for the Data Abort vector fetch.
- The processor updates the FAR on debug exception entry because of watchpoints, although this is architecturally Unpredictable. It is set to the *Modified Virtual Address* (MVA) that triggered the watchpoint.

not sure what they mean by
arch unpredict --- can we not
depend on this on our
specific arm chip?

- The WFAR is set whenever a watchpoint debug event generates either a Debug exception or Debug state entry. It is set to the VA of the instruction that caused the Watchpoint debug event, plus an offset dependent on the processor state. These offsets are the same as the ones that Table 13-25 on page 13-39 lists.

  Table 13-23 lists the setting of CP15 registers on debug events.

Table 13-23 Setting of CP15 registers on debug events

	Debug exception taken because of:		**Debug state entry because of:**	
Register	**A breakpoint, software breakpoint, or vector catch debug event**	**A watchpoint debug event**	**A debug event other than a watchpoint**	**A watchpoint debug event**
IFSR	Cause of Prefetch Abort exception handler entry	Unchanged	Unchanged	Unchanged
DFSR	Unchanged	Cause of Data Abort exception handler entry	Unchanged	Unchanged
FAR	Unchanged	Watchpointed address	Unchanged	Unchanged
WFAR	Unchanged	Address of the instruction causing the watchpoint debug event	Unchanged	Address of the instruction causing the watchpoint debug event

You must take care when setting a breakpoint or software breakpoint debug event inside the Prefetch Abort or Data Abort exception handlers, or when setting a watchpoint debug event on a data address that might be accessed by any of these handlers. These debug events overwrite the R14_abt, SPRS_abt and the CP15 registers listed in this section, leading to an unpredictable software behavior if the handlers did not have the chance of saving the registers.

13.9 Debug exception

When a Software debug event occurs and Monitor debug-mode is selected and enabled and the core is in a state that permits debug then a Debug exception is taken. Prefetch Abort and Data Abort Vector catch debug events are ignored though. Unlinked context ID and any IMVA mismatch breakpoint debug events are also ignored if the processor is running in a privileged mode and Monitor debug-mode is selected and enabled. If the cause of the Debug exception is a watchpoint debug event, the processor performs the following actions:

- The DSCR[5:2] method of entry bits are set to indicate that a watchpoint occurred.
- The CP15 DFSR, FAR, and WFAR are set as *Effect of a debug event on CP15 registers* on page 13-34 describes.
- The same sequence of actions as in a Data Abort exception is performed. This includes setting the R14_abt, base register and destination registers to the same values as if this was a Data Abort.

how to handle watchpoint
or breakpoint exception

The Data Abort handler is responsible for checking the DFSR bit to determine if the routine entry was caused by a debug exception or a Data Abort exception. On entry:

1. It must first check for the presence of a debug monitor target.
2. If present, the handler must disable the active watchpoints. This is necessary to prevent corruption of the FAR because of an unexpected watchpoint debug event when servicing a Data Abort exception.
3. If the cause is a Debug exception the Data Abort handler branches to the debug monitor target.

   Note

   - the watchpointed address can be found in the FAR
   - the address of the instruction that caused the watchpoint debug event can be found in the WFAR
   - the address of the instruction to restart at plus `0x08` can be found in the R14_abt register.

If the cause of the Debug exception is a breakpoint, software breakpoint or vector catch debug event, the processor performs the following actions:

- the DSCR[5:2] method of entry bits are set appropriately
- the CP15 IFSR register is set as *Effect of a debug event on CP15 registers* on page 13-34 describes.
- the same sequence of actions as in a Prefetch Abort exception is performed.

The Prefetch Abort handler is responsible for checking the IFSR bits to find out if the routine entry is caused by a Debug exception or a Prefetch Abort exception. If the cause is a Debug exception it branches to the debug monitor target.

Note

The address of the instruction causing the Software debug event plus `0x04` can be found in the R14_abt register.

Table 13-24 on page 13-36 lists the values in the link register after exceptions.

Table 13-24 Values in the link register after exceptions

note differences!

Cause of fault	ARM	Thumb	Jazelle	Return address (RA[a]) meaning
Breakpoint	RA+4	RA+4	RA+4	Breakpointed instruction address
Watchpoint	RA+8	RA+8	RA+8	Address of the instruction where the execution resumes, a number of instructions after the one that hit the watchpoint
BKPT instruction	RA+4	RA+4	RA+4	BKPT instruction address
Vector catch	RA+4	RA+4	RA+4	Vector address
Prefetch Abort	RA+4	RA+4	RA+4	Address of the instruction where the execution resumes
Data Abort	RA+8	RA+8	RA+8	Address of the instruction where the execution resumes

[a] This is the address of the instruction that the processor first executes on Debug state exit. Watchpoints can be imprecise. RA is not the address of the instruction immediately after the one that hit the watchpoint, the processor might stop a number of instructions later. The address of the instruction that hit the watchpoint is in the CP15 WFAR.

~~13.10~~ Debug state

When the conditions in *Behavior of the processor on debug events* on page 13-33 are met then the processor switches to Debug state. While in Debug state, the processor behaves as follows:

- The DSCR[0] core halted bit is set.
- The **DBGACK** signal is asserted, see *External signals* on page 13-52.
- The DSCR[5:2] method of entry bits are set appropriately.
- The CP15 IFSR, DFSR, FAR, and WFAR registers are set as *Effect of a debug event on CP15 registers* on page 13-34 describes.
- The processor is halted. The pipeline is flushed and no instructions are fetched.
- The processor does not change the execution mode. The CPSR is not altered.
- The DMA engine keeps on running. The DBGTAP debugger can stop it and restart it using CP15 operations if it has permission to do so. See Chapter 7 *Level One Memory System* for details.
- Interrupts and exceptions are treated as *Interrupts* on page 13-39 and *Exceptions* on page 13-39 describe.
- Software debug events are ignored.
- The external debug request signal is ignored.
- Debug state entry request commands are ignored.
- There is a mechanism, using the Debug Test Access Port, where the core is forced to execute an ARM state instruction. This mechanism is enabled using DSCR[13] execute ARM instruction enable bit.
- The core executes the instruction as if it is in ARM state, regardless of the actual value of the T and J bits of the CPSR.
- Any instruction issued in Debug state that puts the processor into a mode or state where debug is not permitted is ignored.
- When in Debug state the CPSR must be modified using the MSR instruction.
- In Debug state MSR can be used to modify the CPSR mode bits from any mode to any mode that is permitted by the debug level set by **SPIDEN** and SUIDEN.

  For example, if **SPIDEN** is set, the CPSR mode bits can be altered to change to Secure Monitor mode from any mode, including all Non-secure modes.

  The CPSR mode can be altered from Non-secure User mode to any Non-secure Privileged mode regardless of the state of **SPIDEN**.
- Instructions that write to the I, F, and A bits of the CPSR are ignored when:
  - debug is only permitted in Non-secure world and in Secure User mode, SPIDEN=0, SUIDEN=1
  - the processor is in Secure user mode
- The MSR instruction can also be used to alter the J and T execution state bits of the CPSR.
- The PC behaves as *Behavior of the PC in Debug state* on page 13-38 describes.

- Instructions that access CP14 registers are always permitted in Debug state. This applies regardless of the debug permissions and the processor mode and state. For example even if:
  - — debug is only permitted in Non-secure world and in Secure User mode, SPIDEN=0, SUIDEN=1
  - — the processor is in Secure user mode
- For CP15 registers in Debug state the processor behaves as follows:
  - — If the debugger is permitted to write to the CPSR mode bits in the current world and change to a privileged mode, then the debugger is permitted to access the CP15 registers of that world. There is no requirement to change to a privileged mode first.
  - — Access to the CP15 registers of that world is then limited to the access granted to any privileged mode in that world.
  - — Any attempts to perform accesses that are not permitted are treated as Undefined Exceptions and cause the sticky Undefined bit to be set in the DSCR.

  For example:

  - — If debug is permitted everywhere, then if the processor is stopped in any Secure mode, including Secure User mode, it has the same access to the Secure banked CP15 registers as any Secure privileged mode. However, if the processor is stopped in a Non-secure mode, including Non-secure User mode, the debugger can only directly access the Non-secure banked CP15 registers, and those CP15 registers, for example NSAC, or bits of CP15 registers, for example the B, FI, L4 and RR bits of the Control Register, that are not banked and are read-only in Non-secure modes are read-only to the debugger. The debugger can write to the CPSR mode bits to switch to Secure Monitor mode, and thereby set or clear the NS bit to read or write all CP15 registers in either bank.
  - — If debug is permitted only in Non-secure state and in Secure User mode, then if the processor is stopped in Secure User mode, it has no privileged access to any CP15 registers. If the processor is stopped in any Non-secure mode, including Non-secure User mode, then it can only access the Non-secure banked CP15 registers, and those CP15 registers or bits of CP15 registers that are not banked and are read-only in Non-secure modes are read-only to the debugger. The debugger cannot write to the mode bits to change the processor into Secure Monitor mode, so cannot access any Secure CP15 registers.
  - — If debug is permitted only in Non-secure state, the processor can only be stopped in Non-secure modes, including Non-secure User mode. It can only access the Non-secure banked CP15 registers, and those CP15 registers or bits of CP15 registers that are not banked and are read-only in Non-secure modes are read-only to the debugger. The debugger cannot write to the mode bits to change the processor into Secure Monitor mode, so cannot access any Secure CP15 registers.
- A DBGTAP debugger can force the processor out of Debug state by issuing a Restart instruction. See Table 14-1 on page 14-6. The Restart command clears the DSCR[1] core restarted flag. When the processor has actually exited Debug state, the DSCR[1] core restarted bit is set and the DSCR[0] core halted bit and **DBGACK** signal are cleared.

13.10.1 Behavior of the PC in Debug state

In Debug state:

- The PC is frozen on entry to Debug state. That is, it does not increment on the execution of ARM instructions. However, branches and instructions that modify the PC directly do update it.

- If the PC is read after the processor has entered Debug state, it returns a value as Table 13-25 lists, depending on the previous state and the type of debug event.
- If a sequence for writing a certain value to the PC is executed while in Debug state, and then the processor is forced to restart, execution starts at the address corresponding to the written value. However, the CPSR has to be set to the return ARM, Thumb, or Jazelle state before the PC is written to, otherwise the processor behavior is Unpredictable.
- If the processor is forced to restart without having performed a write to the PC, the restart address is Unpredictable.
- If the PC or CPSR are written to while in Debug state, subsequent reads to the PC return an Unpredictable value.
- The MSR instruction has an Unpredictable effect on the PC so the PC must be written before leaving Debug state.
- If a conditional branch is executed and it fails its condition code, an Unpredictable value is written to the PC.

Table 13-25 lists the read PC value after Debug state entry for different debug events.

Table 13-25 Read PC value after Debug state entry

Debug event	ARM	Thumb	Jazelle	Return address (RA[a]) meaning
Breakpoint	RA+8	RA+4	RA	Breakpointed instruction address
Watchpoint	RA+8	RA+4	RA	Address of the instruction where the execution resumes, several instructions after the one that hit the watchpoint
BKPT instruction	RA+8	RA+4	RA	BKPT instruction address
Vector catch	RA+8	RA+4	RA	Vector address
External debug request signal activation	RA+8	RA+4	RA	Address of the instruction where the execution resumes
Debug state entry request command	RA+8	RA+4	RA	Address of the instruction where the execution resumes

[a] This is the address of the instruction that the processor first executes on Debug state exit. Watchpoints can be imprecise. RA is not the address of the instruction immediately after the one that hit the watchpoint, the processor might stop a number of instructions later. The address of the instruction that hit the watchpoint is in the CP15 WFAR.

13.10.2 Interrupts

Interrupts are ignored regardless of the value of the I and F bits of the CPSR, although these bits are not changed because of the Debug state entry.

13.10.3 Exceptions

Exceptions are handled as follows while in Debug state:

Reset This exception is taken as in a normal processor state, ARM, Thumb, or Jazelle. This means the processor leaves Debug state as a result of the system reset.

Prefetch Abort

This exception cannot occur because no instructions are prefetched while in Debug state.

Debug This exception cannot occur because software debug events are ignored while in Debug state.

SVC The instruction is ignored.

SMC The instruction is ignored.

Undefined Exception

When an Undefined exception occurs in Debug state, the behavior of the core is as follows:

- PC, CPSR, SPSR_und, R14_und and DSCR[5:2], method of entry bits, are unchanged.
- The processor remains in Debug state.
- DSCR[8], sticky undefined bit, is set.

Precise Data abort

When a precise Data Abort occurs in Debug state the behavior of the core is as follows:

- PC, CPSR, SPSR_abt, R14_abt and DSCR [5:2], method of entry bits, are unchanged
- the processor remains in Debug state
- DSCR[6], sticky precise data abort bit, is set
- DFSR and FAR are set.

Imprecise Data Abort

When an imprecise Data Abort is detected in Debug state, the behavior of the core is as follows, regardless of the setting of the CPSR A bit:

- PC, CPSR, SPSR_abt, R14_abt and DSCR[5:2], method of entry bits, are unchanged.
- The processor remains in Debug state.
- DSCR[7], sticky imprecise data abort bit, is set.
- The imprecise Data Abort is not taken, so DFSR is not set and the FAR is not updated.

Note

The DFSR and FAR that are updated depends on if the core is in a Secure or Non-secure state. The registers that can be read in Debug state depends on the current setting of the NS bit. The DFSR and FAR are always updated for precise data aborts in Debug state even when the processor is in Secure User mode, and SPIDEN is not set. In such circumstances the debugger has no access to DFSR and FAR to restore their values.

Imprecise Data Aborts in detail

The processor takes imprecise data abort exceptions when:

- an imprecise data abort is pending
- the A bit in the CPSR is not set
- the processor is not in Debug state.

On entry to Debug state, DSCR[19] is normally zero. The debugger must issue a Data Memory Barrier operation to flush all pending memory operations to the system. Once these operations have completed, the processor sets DSCR[19]. If any of these operations cause imprecise data aborts, the processor latches the abort and its type until the processor leaves Debug state, in the same way as if an imprecise data abort is detected in normal operation when the A bit in the CPSR is set. The aborts are not taken immediately.

When the processor sets this bit, any memory accesses from Debug state that cause imprecise data aborts cause DSCR[7], sticky imprecise data abort, to be set, but are otherwise discarded. The cause and type of the abort are not recorded. In particular, if an abort is still latched from the initial Data Memory Barrier that was completed on entry to Debug state, it is not overwritten by the new abort. Following writes to memory by the debugger it issues a Data Memory Barrier operation to ensure imprecise data aborts are detected.

Before exit from Debug state, a debugger must issue a Data Memory Barrier operation. On exit from Debug state, DSCR[19] is cleared by the processor.

If an imprecise data abort has occurred during the period between entry to Debug state and the when the processor set DSCR[19], it is taken by the processor on exit from Debug state, providing the A bit in the CPSR is not set. If the A bit in the CPSR is set, it is pended until the A bit in the CPSR is cleared, as for normal operation.

Table 13-26 lists an example sequence of a memory operation executed in normal operation that eventually causes an imprecise abort when the processor is in Debug state. In addition, a memory operation issued by the debugger in Debug state causes a second imprecise abort that is ignored by the processor, apart from the sticky imprecise data abort bit being set. Throughout the example the A bit in the CPSR is clear.

Table 13-26 Example memory operation sequence

	Operation	Result	Debug state?	DCSR[19]	DCSR[7]	Abort latched?	Abort taken?
1	Memory write	Buffered operation	No	0	0		
2	Debug exception	Enters Debug state	Yes	0	0		
3	Data Memory Barrier	Buffered operation flushed - imprecise data abort	Yes	0	1[a]	Yes	No[b]
4		Processor sets DSCR[19]	Yes	1	1		
5	DSCR read	Clears sticky bits	Yes	1	0		
6	Memory write	Buffered operation	Yes	1	0		
7	Data Memory Barrier	Buffered operation flushed - imprecise data abort	Yes	1	1	No[c]	No
8	DSCR read	Clears sticky bits	Yes	1	0		
9	Exit Debug state	Processor clears DSCR[19]	No	0	0		Yes[d](d)

a. The sticky imprecise data abort bit is set because an imprecise data abort was signalled in Debug state.

b. Abort is not taken because the processor is in Debug state.

c. Abort is not latched because DSCR[19] is set.

d. The previous abort latched on row (3) is taken, now the processor has left Debug state and the A bit in the CPSR is not set.

~~13.11~~ Debug communications channel

There are two ways that a DBGTAP debugger can send data to or receive data from the core:

- The debug communications channel, when the core is not in Debug state. It is defined as the set of resources used for communicating between the DBGTAP debugger and a piece of software running on the core.
- The mechanism for forcing the core to execute ARM instructions, when the core is in Debug state. For details see *Executing instructions in Debug state* on page 14-21.

At the core side, the debug communications channel resources are:

- CP14 Debug Register c5, DTR. Data coming from a DBGTAP debugger can be read by an MRC or STC instruction addressed to this register. The core can write to this register any data intended for the DBGTAP debugger, using an MCR or LDC instruction. Because the DTR comprises both a read, rDTR, and a write portion, wDTR, a data item written by the core can be held in this register at the same time as one written by the DBGTAP debugger.
- Some flags and control bits of CP14 Debug Register c1, DSCR:
  - — User mode access to comms channel disable, DSCR[12]. If this bit is set, only privileged software is able to access the debug communications channel. That is, access the DSCR and the DTR.
  - — wDTRfull flag, DSCR bit 29. When clear, this flag indicates to the core that the wDTR is ready to receive data. It is automatically cleared on reads of the wDTR by the DBGTAP debugger, and is set on writes by the core to the same register. If this bit is set and the core attempts to write to the wDTR, the register contents are overwritten and the wDTRfull flag remains set.
  - — rDTRfull flag, DSCR bit 30. When set, this flag indicates to the core that there is data available to read at the rDTR. It is automatically set on writes to the rDTR by the DBGTAP debugger, and is cleared on reads by the core of the same register.

Monitor debug-mode debugging on page 14-42 describes the DBGTAP debugger side of the debug communications channel.

13.12 Debugging in a cached system

Debugging must be non-invasive in a cached system. In processor based systems, you can preserve the contents of the cache so the state of the target application is not altered, and to maintain memory coherency during debugging.

To preserve the contents of the level one cache, you can disable the Instruction Cache and Data Cache line fills so read misses from main memory do not update the caches. You can put the caches in this mode by programming the operation of the caches during debug using CP14 c10. See *CP14 c10, Debug State Cache Control Register* on page 13-23. This facility is accessible from both the core and DBGTAP debugger sides.

In Debug state, the caches behave as follows, for memory coherency purposes:

- Cache reads behave as for normal operation.
- Writes are covered in *Data cache writes*.
- ARMv6 includes CP15 instructions for cleaning and invalidating the cache content, See *c7, Cache operations* on page 3-69. These instructions enable you to reset the processor memory system to a known safe state, and are accessible from both the core and the DBGTAP debugger side.

When the processor is in Secure User mode and **SPIDEN** is not asserted, only the User mode CP15 registers are accessible with the exception of Invalidate Instruction Cache Range and Flush Entire BTAC that are always accessible in Debug state.

13.12.1 Data cache writes

The problem with Data Cache writes is that, while debugging, you might want to write some instructions to memory, either some code to be debugged or a BKPT instruction. This poses coherency issues on the Instruction Cache. In processor based systems, CP14 c10, the Debug State Cache Control Register, enables you to use the following features:

- You can put the processor in a state where data writes work as if the cache is enabled and every region of memory is Write-Through. See *CP14 c10, Debug State Cache Control Register* on page 13-23.
- ARMv6 architecture provides CP15 instructions for invalidating the Instruction Cache, specifically Invalidate Instruction Cache range and Flush Entire Branch Target Address Cache, that *c7, Cache operations* on page 3-69 describes, to ensure that, after a write, there are no out-of-date words in the Instruction Cache.

~~13.13~~ Debugging in a system with TLBs

interesting, but
we don't worry about
this for the moment.

Debugging in a system with TLBs has to be as non-invasive as possible. There has to be a way to put the TLBs in a state where their contents are not affected by the debugging process. The processor enables you to put the TLBs in this mode using CP14 c11. See *CP14 c11, Debug State MMU Control Register* on page 13-23.

13.14 Monitor debug-mode debugging

Monitor debug-mode debugging is essential in real-time systems when the integer core cannot be halted to collect information. Engine controllers and servo mechanisms in hard drive controllers are examples of systems that might not be able to stop the code without physically damaging components. These are typical systems that can be debugged using Monitor debug-mode.

better make sure your handler is fast enough!

For situations that can only tolerate a small intrusion into the instruction stream, Monitor debug-mode is ideal. Using this technique, code can be suspended with an exception long enough to save off state information and important variables. The code continues when the exception handler is finished. The IFSR and DFSR indicate whether a debug exception has occurred, and if it has, the *Method Of Entry* (MOE) bits in the DSCR can be read to determine what caused the exception.

When in Monitor debug-mode, all breakpoint and watchpoint registers can be read and written with MRC and MCR instructions from a privileged processing mode.

13.14.1 Entering the debug monitor target

No debug-mode is the selected default by on power-on reset. Monitor debug-mode must be selected after reset by setting DSCR[15]. See *CP14 c1, Debug Status and Control Register (DSCR)* on page 13-7. When a software debug event occurs, as *Software debug event* on page 13-32 describes, and Monitor debug-mode is selected and enabled, then a Debug exception is taken, although Prefetch Abort and Data Abort vector catch debug events are ignored. *Debug exception* on page 13-35 describes debug exception entry. The Prefetch Abort handler can check the IFSR, and the Data Abort handler can check the DFSR, to find out the caused of the exception. If the cause was a Debug exception, the handler branches to the debug monitor target. When the debug monitor target is running, it can determine and modify the processor state and new software debug events can be programmed.

13.14.2 Setting breakpoints, watchpoints, and vector catch debug events

useful summary / cookbook.

When the debug monitor target is running, breakpoints, watchpoints, and vector catch debug events can be set. This can be done by executing MCR instructions to program the appropriate CP14 debug registers. The debug monitor target can only program these registers if the processor is in a privileged mode and Monitor debug-mode is selected and enabled, see *Debug Status and Control Register bit field definitions* on page 13-8.You can program a vector catch debug event using CP14 Debug Vector Catch Register.

You can program a breakpoint debug event using CP14 Debug Breakpoint Value Registers and CP14 Debug Breakpoint Control Registers, see *CP14 c64-c69, Breakpoint Value Registers (BVR)* on page 13-16 and *CP14 c80-c85, Breakpoint Control Registers (BCR)* on page 13-17.You can program a watchpoint debug event using CP14 Debug Watchpoint Value Registers and CP14 Debug Watchpoint Control Registers, see *CP14 c96-c97, Watchpoint Value Registers (WVR)* on page 13-20, and *CP14 c112-c113, Watchpoint Control Registers (WCR)* on page 13-21.

Setting a simple breakpoint on an IMVA

You can set a simple breakpoint on an IMVA as follows:

1. Read the BCR.
2. Clear the BCR[0] enable breakpoint bit in the read word and write it back to the BCR. Now the breakpoint is disabled.

in general, don't want to take an exception while setting a new one.

3. Write the IMVA to the BVR register.

4. Write to the BCR with its fields set as follows:
    - BCR[22:21] meaning of BVR bit set to b00 or b10, to indicate that the value loaded into BVR is to be compared against the IMVA bus as a match or mismatch.
    - BCR[20] enable linking bit cleared, to indicate that this breakpoint is not to be linked.
    - BCR [15:14] Secure access BCR field as required.
    - BCR[8:5] byte address select BCR field as required.
    - BCR[2:1] supervisor access BCR field as required.
    - BCR[0] enable breakpoint bit set.

Note

Any BVR can be compared against the IMVA bus.

Setting a simple breakpoint on a context ID value

A simple breakpoint on a context ID value can be set, using one of the context ID capable BRPs, as follows:

1. Read the BCR.

2. Clear the BCR[0] enable breakpoint bit in the read word and write it back to the BCR. Now the breakpoint is disabled.

3. Write the context ID value to the BVR register.

4. Write to the BCR with its fields set as follows:
    - BCR[22:21] meaning of BVR bit set to b01, to indicate that the value loaded into BVR is to be compared against the CP15 Context Id Register c13.
    - BCR[20] enable linking bit cleared, to indicate that this breakpoint is not to be linked.
    - BCR [15:14] Secure access BCR field as required.
    - BCR[8:5] byte address select BCR field set to b1111.
    - BCR[2:1] supervisor access BCR field as required.
    - BCR[0] enable breakpoint bit set.

Note

Any BVR can be compared against the IMVA bus.

Setting a linked breakpoint

In the following sequence b is any of the breakpoint registers pairs with context ID comparison capability, and a is any of the implemented breakpoints different from b. You can link IMVA holding and contextID-holding breakpoints register pairs as follows:

1. Read the BCRa and BCRb.

2. Clear the BCRa[0] and BCRb[0] enable breakpoint bits in the read words and write them back to the BCRs. Now the breakpoints are disabled.

3. Write the IMVA to the BVRa register.

4. Write the context ID to the BVRb register.

5. Write to the BCRb with its fields set as follows:
   - BCRb[22:21] meaning of BVR bit set to b01, to indicate that the value loaded into BVRb is to be compared against the CP15 context ID register 13
   - BCRb[20] enable linking bit, set
   - BCR [15:14] Secure access set to b00.
   - BCRb[8:5] byte address select set to b1111
   - BCRb[2:1] supervisor access set to b11
   - BCRb[0] enable breakpoint bit set.

6. Write to the BCRa with its fields set as follows:
   - BCRa[22:21] meaning of BVR bit set to b00 or b10, to indicate that the value loaded into BVRa is to be compared against the IMVA bus as a match or mismatch
   - BCRa[20] enable linking bit set, to link this BRP with the one indicated by BCRa[19:16], BRPb in this example
   - BCR [15:14] Secure access as required.
   - binary representation of b into BCR[9:6] linked BRP field
   - BCRa[8:5] byte address select field as required
   - BCRa[2:1] supervisor access field as required
   - BCRa[0] enable breakpoint set.

Setting a simple watchpoint

You can set a simple watchpoint as follows:

1. Read the WCR.

2. Clear the WCR[0] enable watchpoint bit in the read word and write it back to the WCR. Now the watchpoint is disabled.

3. Write the DMVA to the WVR register.

4. Write to the WCR with its fields set as follows:
   - WCR[20] enable linking bit cleared, to indicate that this watchpoint is not to be linked
   - WCR byte address select, load/store access, Secure access field, and supervisor access fields as required
   - WCR[0] enable watchpoint bit set.

Note

Any WVR can be compared against the DMVA bus.

Setting a linked watchpoint

In the following sequence b is any of the BRPs with context ID comparison capability. You can use any of the WRPs. You can link WRPs and contextID-holding BRPs as follows:

1. Read the WCR and BCRb.

2. Clear the WCR[0] Enable Watchpoint and the BCRb[0] Enable breakpoint bits in the read words and write them back to the WCR and BCRb. Now the watchpoint and the breakpoint are disabled.

3. Write the DMVA to the WVR register.

4. Write the context ID to the BVRb register.

5. Write to the WCR with its fields set as follows:
   - WCR[20] enable linking bit set, to link this WRP with the BRP indicated by WCR[19:16], BRPb in this example
   - Binary representation of b into WCR[19:6] linked BRP field
   - WCR byte address select, load/store access, Secure access field, and supervisor access fields as required
   - WCR[0] enable watchpoint bit set.

6. Write to the BCRb with its fields set as follows:
   - BCRb[22:21] meaning of BVR bit set to b01, to indicate that the value loaded into BVRb is to be compared against the CP15 Context ID Register.
   - BCRb[20] enable linking bit, set
   - BCR [15:14] Secure access set to b00
   - BCRb[8:5] byte address select set to b1111
   - BCRb[2:1] supervisor access set to b11
   - BCRb[0] enable breakpoint bit set.

13.14.3 Setting software breakpoint debug events (BKPT)

this uses the bkpt instruction.

To set a software breakpoint on a particular virtual address, the debug monitor target must perform the following steps:

1. Read memory location and save actual instruction.

2. Write BKPT instruction to the memory location.

3. Read memory location again to check that the BKPT instruction has been written.

4. If it has not been written, determine the reason.

Note

Cache coherency issues might arise when writing a BKPT instruction. See *Debugging in a cached system* on page 13-43.

13.14.4 Using the debug communications channel

To read a word sent by a DBGTAP debugger:

1. Read the DSCR register.

2. If DSCR[30] rDTRfull flag is clear, then go to 1.

3. Read the word from the rDTR, CP14 Debug Register c5.

To write a word for a DBGTAP debugger:

1. Read the DSCR register.

2. If DSCR[29] wDTRfull flag is set, then go to 1.
3. Write the word to the wDTR, CP14 Debug Register c5.

13.15 Halting debug-mode debugging

Halting debug-mode is used to debug the processor using external hardware connected to the DBGTAP. The external hardware provides an interface to a DBGTAP debugger application. You can only select Halting debug-mode by setting the halt bit, bit [14], of the DSCR. You can only write to it through the Debug Test Access Port. See Chapter 14 *Debug Test Access Port*.

In Halting debug-mode the processor stops executing instructions and enters Debug state if one of the following events occurs:

- a breakpoint hits
- a watchpoint hits
- a BKPT instruction is executed
- the **EDBGRQ** signal is asserted
- a Halt instruction has been scanned into the DBGTAP instruction register
- an vector catch occurs.

When the processor is in Debug state, you control it by sending instructions to the integer core through the DBGTAP. This enables you to scan any valid instruction into the processor. The effect of the instruction on the integer core is as if it was executed under normal operation. A register to transfer data between CP14 and the DBGTAP debugger is also accessible through the DBGTAP.

A DBGTAP Restart instruction restarts the integer core.

13.15.1 Entering Debug state

When a debug event occurs and Halting debug-mode is selected and enabled and the core is in a state when debug is permitted then the processor enters Debug state as defined in *Debug state* on page 13-37.When the core is in Debug state, the DBGTAP debugger can determine and modify the processor state and new debug events can be programmed.

13.15.2 Exiting Debug state

You can force the processor out of Debug state using the DBGTAP Restart instruction. See *Exiting Debug state* on page 14-5. The DSCR[1] core restarted bit indicates if the core has already returned to normal operation.

13.15.3 Programming debug events

The following sections describe operations you require for Halting debug-mode debugging :

- *Setting breakpoints, watchpoints, and vector catch debug events*
- *Setting software breakpoints (BKPT)* on page 13-51.

Setting breakpoints, watchpoints, and vector catch debug events

For setting breakpoints, watchpoints, and vector catch debug events when in Halting debug-mode, the debug host has to use the same CP14 debug registers and the same sequence of operations as in Monitor debug-mode debugging. See *Setting breakpoints, watchpoints, and vector catch debug events* on page 13-45. The only difference is that the CP14 debug registers are accessed using the DBGTAP scan chains, see *The DBGTAP port and debug registers* on page 14-6.