Feature Summary

- Small area, high clock frequency.
- 32-bit load/store AVR32A RISC architecture.
- 15 general-purpose 32-bit registers.
- 32-bit Stack Pointer, Program Counter and Link Register reside in register file.
- Fully orthogonal instruction set.
- Pipelined architecture allows one instruction per clock cycle for most instructions.
- Byte, half-word, word and double word memory access.
- Fast interrupts and multiple interrupt priority levels.
- Privileged and unprivileged modes enabling efficient and secure Operating Systems.
- Optional MPU allows for operating systems with memory protection.
- Innovative instruction set together with variable instruction length ensuring industry leading code density.
- DSP extention with saturating arithmetic, and a wide variety of multiply instructions.
- Memory Read-Modify-Write instructions.
- Optional advanced On-Chip Debug system.
- FlashVault™ support through Secure state for executing trusted code alongside nontrusted code on the same CPU.
- Optional floating-point hardware.

AVR32UC

Technical Reference Manual

32002F–03/2010

1. Introduction

AVR32 is a new high-performance 32-bit RISC microprocessor core, designed for cost-sensitive embedded applications, with particular emphasis on low power consumption and high code density. In addition, the instruction set architecture has been tuned to allow for a variety of microarchitectures, enabling the AVR32 to be implemented as low-, mid- or high-performance processors. AVR32 extends the AVR family into the world of 32- and 64-bit applications.

1.1 The AVR family

The AVR family was launched by Atmel in 1996 and has had remarkable success in the 8-and 16-bit flash microcontroller market. AVR32 is complements the current AVR microcontrollers. Through the AVR32 family, the AVR is extended into a new range of higher performance applications that is currently served by 32- and 64-bit processors

To truly exploit the power of a 32-bit architecture, the new AVR32 architecture is not binary compatible with earlier AVR architectures. In order to achieve high code density, the instruction format is flexible providing both compact instructions with 16 bits length and extended 32-bit instructions. While the instruction length is only 16 bits for most instructions, powerful 32-bit instructions are implemented to further increase performance. Compact and extended instructions can be freely mixed in the instruction stream.

1.2 The AVR32 Microprocessor Architecture

The AVR32 is a new innovative microprocessor architecture. It is a fully synchronous synthesisable RTL design with industry standard interfaces, ensuring easy integration into SoC designs with legacy intellectual property (IP). Through a quantitative approach, a large set of industry recognized benchmarks has been compiled and analyzed to achieve the best code density in its class of microprocessor architectures. In addition to lowering the memory requirements, a compact code size also contributes to the core’s low power characteristics. The processor supports byte and half-word data types without penalty in code size and performance.

Memory load and store operations are provided for byte, half-word, word and double word data with automatic sign- or zero extension of half-word and byte data. The C-compiler is closely linked to the architecture and is able to exploit code optimization features, both for size and speed.

In order to reduce code size to a minimum, some instructions have multiple addressing modes. As an example, instructions with immediates often have a compact format with a smaller immediate, and an extended format with a larger immediate. In this way, the compiler is able to use the format giving the smallest code size.

Another feature of the instruction set is that frequently used instructions, like add, have a compact format with two operands as well as an extended format with three operands. The larger format increases performance, allowing an addition and a data move in the same instruction in a single cycle.

Load and store instructions have several different formats in order to reduce code size and speed up execution:

- Load/store to an address specified by a pointer register
- Load/store to an address specified by a pointer register with postincrement
- Load/store to an address specified by a pointer register with predecrement
- Load/store to an address specified by a pointer register with displacement

- Load/store to an address specified by a small immediate (direct addressing within a small page)
- Load/store to an address specified by a pointer register and an index register.

The register file is organized as 16 32-bit registers and includes the Program Counter, the Link Register, and the Stack Pointer. In addition, one register is designed to hold return values from function calls and is used implicitly by some instructions.

The AVR32 core defines several micro architectures in order to capture the entire range of applications. The microarchitectures are named AVR32A, AVR32B and so on. Different microarchitectures are suited to different end applications, allowing the designer to select a microarchitecture with the optimum set of parameters for a specific application.

1.3 Exceptions and Interrupts

The AVR32 incorporates a powerful exception handling scheme. The different exception sources, like Illegal Op-code and external interrupt requests, have different priority levels, ensuring a well-defined behavior when multiple exceptions are received simultaneously. Additionally, pending exceptions of a higher priority class may preempt handling of ongoing exceptions of a lower priority class. Each priority class has dedicated registers to keep the return address and status register thereby removing the need to perform time-consuming memory operations to save this information.

There are four levels of external interrupt requests, all executing in their own context. An interrupt controller does the priority handling of the external interrupts and provides the prioritized interrupt vector to the processor core.

1.4 Java Support

Some AVR32 implementations provide Java hardware acceleration. To reduce gate count, AVR32UC does not implement any such hardware.

1.5 FlashVault

Revision 3 of the AVR32 architecture introduced a new CPU state called Secure State. This state is instrumental in the new security technology named FlashVault. This innovation allows the on-chip flash and other memories to be partially programmed and locked, creating a safe on-chip storage for secret code and valuable software intellectual property. Code stored in the FlashVault will execute as normal, but reading, copying or debugging the code is not possible. This allows a device with FlashVault code protection to carry a piece of valuable software such as a math library or an encryption algorithm from a trusted location to a potentially untrustworthy partner where the rest of the source code can be developed, debugged and programmed.

1.6 Microarchitectures

The AVR32 architecture defines different microarchitectures, AVR32A and AVR32B. This enables implementations that are tailored to specific needs and applications. The microarchitectures provide different performance levels at the expense of area and power consumption.

The AVR32A microarchitecture is targeted at cost-sensitive, lower-end applications like smaller microcontrollers. This microarchitecture does not provide dedicated hardware registers for shadowing of register file registers in interrupt contexts. Additionally, it does not provide hardware registers for the return address registers and return status registers. Instead, all this information is stored on the system stack. This saves chip area at the expense of slower interrupt handling.

Upon interrupt initiation, registers R8-R12 are automatically pushed to the system stack. These registers are pushed regardless of the priority level of the pending interrupt. The return address and status register are also automatically pushed to stack. The interrupt handler can therefore use R8-R12 freely. Upon interrupt completion, the old R8-R12 registers and status register are restored, and execution continues at the return address stored popped from stack.

The stack is also used to store the status register and return address for exceptions and *scall*. Executing the *rete* or *rets* instruction at the completion of an exception or system call will pop this status register and continue execution at the popped return address.

1.7 The AVR32UC architecture

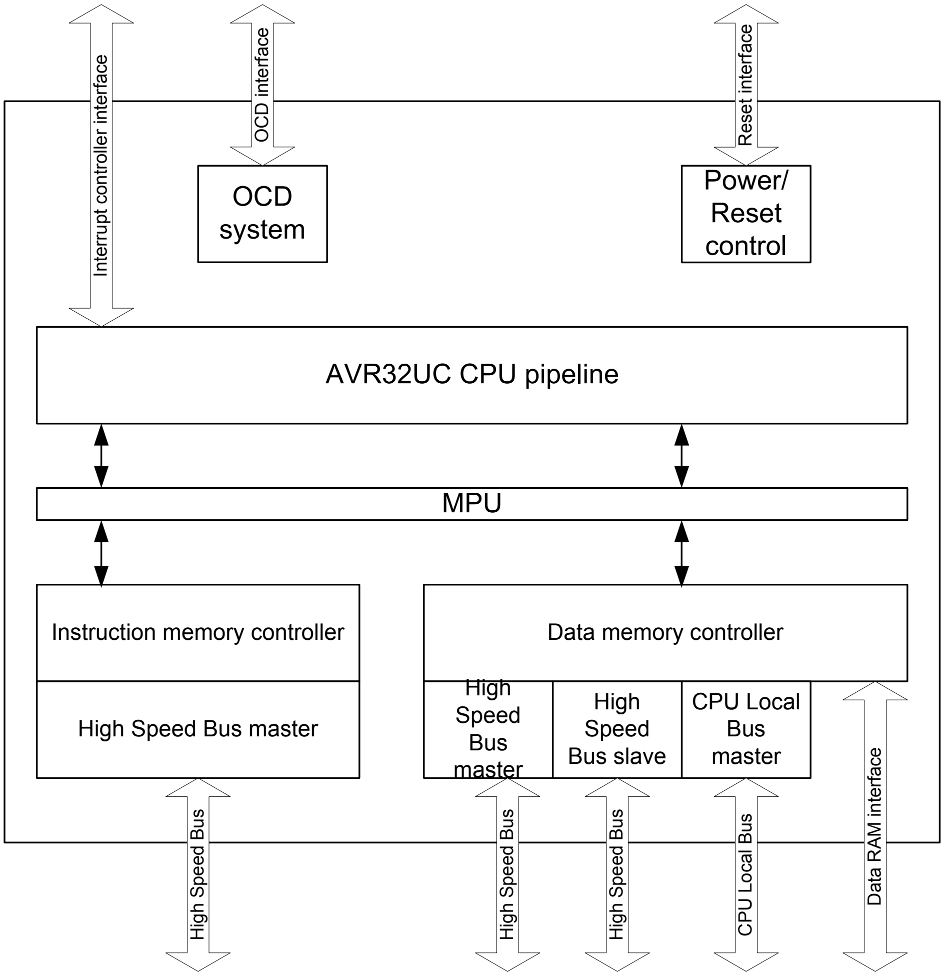
The first implementation of the AVR32A architecture is called AVR32UC. This implementation targets low- and medium-performance applications, and provides an optional, advanced OCD system, no data or instruction caches, and an optional Memory Protection Unit (MPU). Java acceleration is not implemented.

AVR32UC provides three memory interfaces, one High Speed Bus (HSB) master for instruction fetch, one HSB bus master for data access, and one HSB slave interface allowing other bus masters to access data RAMs internal to the CPU. Keeping data RAMs internal to the CPU allows fast access to the RAMs, reduces latency and guarantees deterministic timing. Also, power consumption is reduced by not needing a full HSB bus access for memory accesses. A dedicated data RAM interface is provided for communicating with the internal data RAMs.

If an optional MPU is present, all memory accesses are checked for privilege violations. If an access is attempted to an illegal memory address, the access is aborted and an exception is taken.

The following figure displays the contents of AVR32UC:

Figure 1-1. Overview of AVR32UC.



1.8 AVR32UC CPU revisions

Three revisions of the AVR32UC CPU currently exist:

- Revision 1 implementing revision 1 of the AVR32 architecture.
- Revision 2 implementing revision 2 of the AVR32 architecture, and with a faster divider.
- Revision 3 implementing revision 3 of the AVR32 architecture, and with optional floating-point hardware.

Revision 2 of the AVR32UC CPU added the following instructions:

- movh Rd, imm
- {add, sub, and, or, eor}{cond4}, Rd, Rx, Ry
- ld.{sb, ub, sh, uh, w}{cond4} Rd, Rp[disp]
- st.{b, h, w}{cond4} Rp[disp], Rs
- rsub{cond4} Rd, imm

Revision 3 of the AVR32UC CPU added the following instructions:

- sscall
- retss
- Floating-point instructions as described in Section 4. on page 40.

Revision 3 of the AVR32UC CPU added the following system registers:

- SS_STATUS
- SS_ADRF, SS_ADRR, SS_ADR0, SS_ADR1
- SS_SP_SYS, SS_SP_APP
- SS_RAR, SS_RSR

Revision 3 of the AVR32UC CPU added the following bit in the status register:

- SS

AVR32UC CPU revision 2 is fully backward-compatible with revision 1, ie. code compiled for revision 1 is binary-compatible with revision 2 CPUs.

AVR32UC CPU revision 3 is fully backward-compatible with revision 1 and 2, ie. code compiled for revision 1 and 2 is binary-compatible with revision 3 CPUs.

The Architecture Revision field in the CONFIG0 system register identifies which architecture revision is implemented in a specific device. The “Processor and Architecture”-chapter of the device datasheet identifies the CPU revision used.

2. Programming Model

This chapter describes the programming model and the set of registers accessible to the user. It also describes the implementation options in AVR32UC.

2.1 Architectural compatibility

AVR32UC is fully compatible with the Atmel AVR32A architecture. AVR32UC devices implementing both revision 2 and revision 3 of the AVR32 Architecture exist. Refer to the device datasheet or the device's CONFIG0 register to determine which architecture revision the device implements.

Architecture revision 3 is fully backwards compatible with revision 2, and additionally implements:

- Secure state with associated programming model
- The automatic clearing of COUNT on COMPARE match is now optional and disabled by setting the NOCOMPRES bit in CPUCR.

2.2 Implementation options

2.2.1 Memory protection

AVR32UC optionally supports an MPU as specified by the AVR32 architecture.

2.2.2 Java support

AVR32UC does not implement Java hardware acceleration.

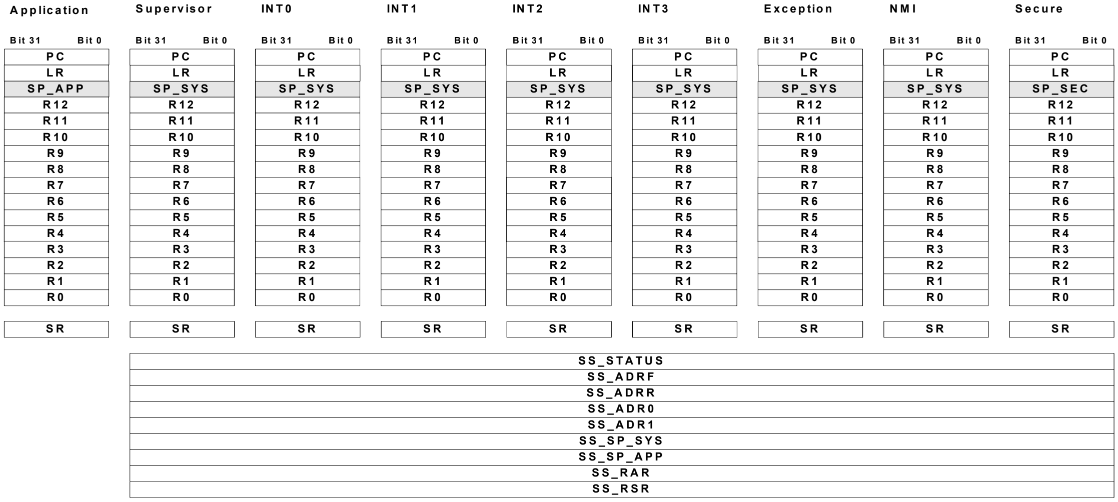
2.2.3 Floating-Point Hardware

AVR32UC optionally supports Floating-Point Hardware implemented as coprocessor instructions.

2.3 Register file configuration

The AVR32A architecture dictates a specific register file implementation, reproduced below. Secure state context and secure state system registers are only available in devices implementing revision 3 of the AVR32 architecture.

Figure 2-1. Register File in AVR32A



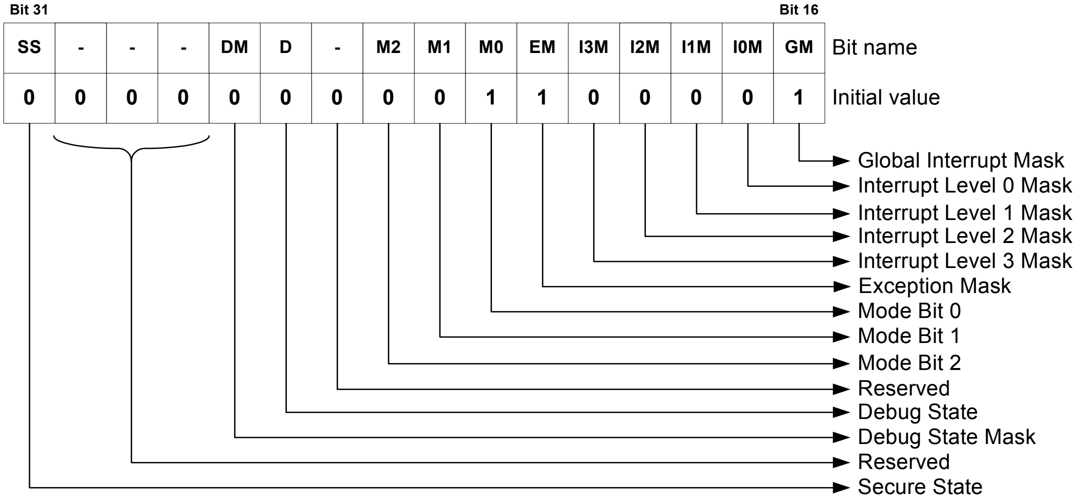
Application	Supervisor	INT0	INT1	INT2	INT3	Exception	NMI	Secure
Bit 31 Bit 0	Bit 31 Bit 0	Bit 31 Bit 0	Bit 31 Bit 0	Bit 31 Bit 0	Bit 31 Bit 0	Bit 31 Bit 0	Bit 31 Bit 0	Bit 31 Bit 0
PC	PC	PC	PC	PC	PC	PC	PC	PC
LR	LR	LR	LR	LR	LR	LR	LR	LR
SP_APP	SP_SYS	SP_SYS	SP_SYS	SP_SYS	SP_SYS	SP_SYS	SP_SYS	SP_SEC
R12	R12	R12	R12	R12	R12	R12	R12	R12
R11	R11	R11	R11	R11	R11	R11	R11	R11
R10	R10	R10	R10	R10	R10	R10	R10	R10
R9	R9	R9	R9	R9	R9	R9	R9	R9
R8	R8	R8	R8	R8	R8	R8	R8	R8
R7	R7	R7	R7	R7	R7	R7	R7	R7
R6	R6	R6	R6	R6	R6	R6	R6	R6
R5	R5	R5	R5	R5	R5	R5	R5	R5
R4	R4	R4	R4	R4	R4	R4	R4	R4
R3	R3	R3	R3	R3	R3	R3	R3	R3
R2	R2	R2	R2	R2	R2	R2	R2	R2
R1	R1	R1	R1	R1	R1	R1	R1	R1
R0	R0	R0	R0	R0	R0	R0	R0	R0
SR	SR	SR	SR	SR	SR	SR	SR	SR

SS_STATUS
SS_ADRF
SS_ADRR
SS_ADR0
SS_ADR1
SS_SP_SYS
SS_SP_APP
SS_RAR
SS_RSR

2.4 The Status Register

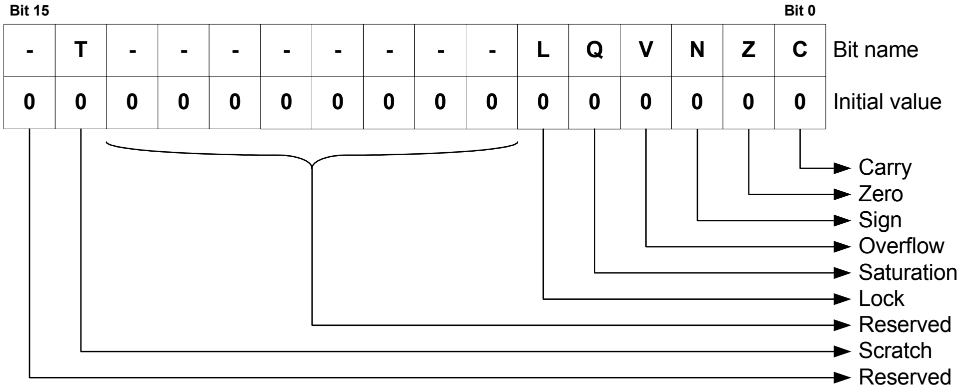
The Status Register (SR) consists of two halfwords, one upper and one lower, see Figure 2-2 on page 8 and Figure 2-3 on page 9. The lower halfword contains the C, Z, N, V and Q flags, as well as the L and T bits, while the upper halfword contains information about the mode and state the processor executes in. The upper halfword can only be accessed from a privileged mode.

Figure 2-2. The Status Register high halfword



Bit 31															Bit 16	
SS	-	-	-	DM	D	-	M2	M1	M0	EM	I3M	I2M	I1M	I0M	GM	Bit name
0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	Initial value

Figure 2-3. The Status Register low halfword



Bit 15															Bit 0	
-	T	-	-	-	-	-	-	-	-	L	Q	V	N	Z	C	Bit name
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Initial value

SS - Secure State

This bit is indicates if the processor is executing in the secure state. Only implemented in devices implementing revision 3 of the AVR32 architecture, set to 0 in older revisions. The bit is initialized in an IMPLEMENTATION DEFINED way at reset. Refer to Section 5. ”Secure State” on page 59 for more information.

DM - Debug State Mask

If this bit is set, the Debug State is masked and cannot be entered. The bit is cleared at reset, and can both be read and written by software.

D - Debug State

The processor is in debug state when this bit is set. The bit is cleared at reset and should only be modified by debug hardware, the *breakpoint* instruction or the *retd* instruction. Undefined behaviour may result if the user tries to modify this bit using other mechanisms.

M2, M1, M0 - Execution Mode

These bits show the active execution mode. The settings for the different modes are shown in Table 2-1 on page 10. M2 and M1 are cleared by reset while M0 is set so that the processor is in supervisor mode after reset. These bits are modified by hardware when initiating interrupt or exception processing. Execution of the *scall*, *rets* or *rete* instructions will also change these bits. Undefined behaviour may result if the user tries to modify these bits using the *mtsr*, *ssrf* or *csrf* instructions. If software needs to change these bits, *scall*, *rets* or *rete* should be used, possibly with prior modifications of the stack, to achieve the desired changes in a safe way. Refer to the AVR32 Architecture Manual for the behaviour of these instructions, note especially how the stack is modified after their execution.

Table 2-1. Mode bit settings

M2	M1	M0	Mode
1	1	1	Non Maskable Interrupt
1	1	0	Exception
1	0	1	Interrupt level 3
1	0	0	Interrupt level 2
0	1	1	Interrupt level 1
0	1	0	Interrupt level 0
0	0	1	Supervisor
0	0	0	Application

EM - Exception mask

When this bit is set, exceptions are masked. Exceptions are enabled otherwise. The bit is automatically set when exception processing is initiated or Debug Mode is entered. Software may clear this bit after performing the necessary measures if nested exceptions should be supported. This bit is set at reset.

I3M - Interrupt level 3 mask

When this bit is set, level 3 interrupts are masked. If I3M and GM are cleared, INT3 interrupts are enabled. The bit is automatically set when INT3 processing is initiated. Software may clear this bit after performing the necessary measures if nested INT3s should be supported. This bit is cleared at reset.

I2M - Interrupt level 2 mask

When this bit is set, level 2 interrupts are masked. If I2M and GM are cleared, INT2 interrupts are enabled. The bit is automatically set when INT3 or INT2 processing is initiated. Software may clear this bit after performing the necessary measures if nested INT2s should be supported. This bit is cleared at reset.

I1M - Interrupt level 1 mask

When this bit is set, level 1 interrupts are masked. If I1M and GM are cleared, INT1 interrupts are enabled. The bit is automatically set when INT3, INT2 or INT1 processing is initiated. Software may clear this bit after performing the necessary measures if nested INT1s should be supported. This bit is cleared at reset.

I0M - Interrupt level 0 mask

When this bit is set, level 0 interrupts are masked. If I0M and GM are cleared, INT0 interrupts are enabled. The bit is automatically set when INT3, INT2, INT1 or INT0 processing is initiated. Software may clear this bit after performing the necessary measures if nested INT0s should be supported. This bit is cleared at reset.

GM - Global Interrupt Mask

When this bit is set, all interrupts are disabled. This bit overrides I0M, I1M, I2M and I3M. The bit is automatically set when exception processing is initiated, Debug Mode is entered, or a Java trap is taken. This bit is automatically cleared when returning from a Java trap. This bit is set after reset.

T - Scratch bit

This bit is not set or cleared implicit by any instruction and the programmer can therefore use this bit as a custom flag to for example signal events in the program. This bit is cleared at reset.

L - Lock flag

Used by the conditional store instruction. Used to support atomical memory access. Automatically cleared by *rete*. This bit is cleared after reset.

Q - Saturation flag

The saturation flag indicates that a saturating arithmetic operation overflowed. The flag is sticky and once set it has to be manually cleared by a *csrf* instruction after the desired action has been taken. See the Instruction set description for details.

V - Overflow flag

The overflow flag indicates that an arithmetic operation overflowed. See the Instruction set description for details.

N - Negative flag

The negative flag is modified by arithmetical and logical operations. See the Instruction set description for details.

Z - Zero flag

The zero flag indicates a zero result after an arithmetic or logic operation. See the Instruction set description for details.

C - Carry flag

The carry flag indicates a carry after an arithmetic or logic operation. See the Instruction set description for details.

2.5 System registers

The system registers are placed outside of the virtual memory space, and are only accessible using the privileged *mfsr* and *mtsr* instructions. Some of the System Registers can be altered automatically by hardware. The table below lists the system registers specified in AVR32UC. The programmer is responsible for maintaining correct sequencing of any instructions following a *mtsr* instruction.

Table 2-2. System Registers

Reg #	Address	Name	Function
0	0	SR	Status Register
1	4	EVBA	Exception Vector Base Address
2	8	ACBA	Application Call Base Address
3	12	CPUCR	CPU Control Register
4	16	ECR	Exception Cause Register
5	20	RSR_SUP	Unused in AVR32UC
6	24	RSR_INT0	Unused in AVR32UC
7	28	RSR_INT1	Unused in AVR32UC

Table 2-2. System Registers (Continued)

Reg #	Address	Name	Function
8	32	RSR_INT2	Unused in AVR32UC
9	36	RSR_INT3	Unused in AVR32UC
10	40	RSR_EX	Unused in AVR32UC
11	44	RSR_NMI	Unused in AVR32UC
12	48	RSR_DBG	Return Status Register for Debug Mode
13	52	RAR_SUP	Unused in AVR32UC
14	56	RAR_INT0	Unused in AVR32UC
15	60	RAR_INT1	Unused in AVR32UC
16	64	RAR_INT2	Unused in AVR32UC
17	68	RAR_INT3	Unused in AVR32UC
18	72	RAR_EX	Unused in AVR32UC
19	76	RAR_NMI	Unused in AVR32UC
20	80	RAR_DBG	Return Address Register for Debug Mode
21	84	JECR	Unused in AVR32UC
22	88	JOSP	Unused in AVR32UC
23	92	JAVA_LV0	Unused in AVR32UC
24	96	JAVA_LV1	Unused in AVR32UC
25	100	JAVA_LV2	Unused in AVR32UC
26	104	JAVA_LV3	Unused in AVR32UC
27	108	JAVA_LV4	Unused in AVR32UC
28	112	JAVA_LV5	Unused in AVR32UC
29	116	JAVA_LV6	Unused in AVR32UC
30	120	JAVA_LV7	Unused in AVR32UC
31	124	JTBA	Unused in AVR32UC
32	128	JBCR	Unused in AVR32UC
33-63	132-252	Reserved	Reserved for future use
64	256	CONFIG0	Configuration register 0
65	260	CONFIG1	Configuration register 1
66	264	COUNT	Cycle Counter register
67	268	COMPARE	Compare register
68	272	TLBEHI	Unused in AVR32UC
69	276	TLBELO	Unused in AVR32UC
70	280	PTBR	Unused in AVR32UC
71	284	TLBEAR	Unused in AVR32UC
72	288	MMUCR	Unused in AVR32UC
73	292	TLBARLO	Unused in AVR32UC

Table 2-2. System Registers (Continued)

Reg #	Address	Name	Function
74	296	TLBARHI	Unused in AVR32UC
75	300	PCCNT	Unused in AVR32UC
76	304	PCNT0	Unused in AVR32UC
77	308	PCNT1	Unused in AVR32UC
78	312	PCCR	Unused in AVR32UC
79	316	BEAR	Bus Error Address Register
80	320	MPUAR0	MPU Address Register region 0
81	324	MPUAR1	MPU Address Register region 1
82	328	MPUAR2	MPU Address Register region 2
83	332	MPUAR3	MPU Address Register region 3
84	336	MPUAR4	MPU Address Register region 4
85	340	MPUAR5	MPU Address Register region 5
86	344	MPUAR6	MPU Address Register region 6
87	348	MPUAR7	MPU Address Register region 7
88	352	MPUPSR0	MPU Privilege Select Register region 0
89	356	MPUPSR1	MPU Privilege Select Register region 1
90	360	MPUPSR2	MPU Privilege Select Register region 2
91	364	MPUPSR3	MPU Privilege Select Register region 3
92	368	MPUPSR4	MPU Privilege Select Register region 4
93	372	MPUPSR5	MPU Privilege Select Register region 5
94	376	MPUPSR6	MPU Privilege Select Register region 6
95	380	MPUPSR7	MPU Privilege Select Register region 7
96	384	MPUCRA	MPU Cacheable Register A
97	388	MPUCRB	MPU Cacheable Register B
98	392	MPUBRA	MPU Bufferable Register A
99	396	MPUBRB	MPU Bufferable Register B
100	400	MPUAPRA	MPU Access Permission Register A
101	404	MPUAPRB	MPU Access Permission Register B
102	408	MPUCR	MPU Control Register
103	412	SS_STATUS	Secure State Status Register
104	416	SS_ADRF	Secure State Address Flash Register
105	420	SS_ADRR	Secure State Address RAM Register
106	424	SS_ADR0	Secure State Address 0 Register
107	428	SS_ADR1	Secure State Address 1 Register
108	432	SS_SP_SYS	Secure State Stack Pointer System Register
109	436	SS_SP_APP	Secure State Stack Pointer Application Register

Table 2-2. System Registers (Continued)

Reg #	Address	Name	Function
110	440	SS_RAR	Secure State Return Address Register
111	444	SS_RSR	Secure State Return Status Register
112-191	448-764	Reserved	Reserved for future use
192-255	768-988	IMPL	IMPLEMENTATION DEFINED
248	992	MSU_ADDRHI	Memory Service Unit Address High Register
249	996	MSU_ADDRLO	Memory Service Unit Address Low Register
250	1000	MSU_LENGTH	Memory Service Unit Length Register
251	1004	MSU_CTRL	Memory Service Unit Control Register
252	1008	MSU_STATUS	Memory Service Unit Status Register
253	1012	MSU_DATA	Memory Service Unit Data Register
254	1016	MSU_TAIL	Memory Service Unit TailRegister
255	1020	Reserved	Reserved for future use

SR- Status Register

The Status Register is mapped into the system register space. This allows it to be loaded into the register file to be modified, or to be stored to memory. The Status Register is described in detail in Section 2.4 ”The Status Register” on page 8.

EVBA - Exception Vector Base Address

This register contains a pointer to the exception routines. All exception routines start at this address, or at a defined offset relative to the address. Special alignment requirements may apply for EVBA, depending on the implementation of the interrupt controller. Exceptions are described in detail in the AVR32 Architecture Manual.

ACBA - Application Call Base Address

Pointer to the start of a table of function pointers. Subroutines can thereby be called by the compact *acall* instruction. This facilitates efficient reuse of code. Keeping this pointer as a register facilitates multiple function pointer tables. ACBA is a full 32 bit register, but the lowest two bits should be written to zero, making ACBA word aligned. Failing to do so may result in erroneous behaviour.

CPUCR - CPU Control Register

Register controlling the configuration and behaviour of the CPU. The following fields are defined:

Table 2-3. CPU control register

Name	**Bit**	**Reset**	**Description**
-	Other	-	Unused. Read as 0. Should be written as 0.
NOCOMP RES	17	0	If set, COUNT is not set on COMPARE match. If cleared, COUNT is cleared on COMPARE match.
LOCEN	16	0	Local Bus Enable. Must be written to 1 to enable the local bus. Any access attempted to the LOCAL section when this bit is cleared will result in a BUS ERROR.

Table 2-3. CPU control register

Name	Bit	Reset	Description
SPL	15:11	16	Slave Pending Limit. The maximum number of clock cycles the slave interface can have a request pending due to the CPU owning the RAMs. After this period, the CPU will lose arbitrartion for the RAM, and the slave access can proceed.
CPL	10:6	16	CPU Pending Limit. The maximum number of clock cycles the CPU can have a request pending due to the slave interface owning the RAMs. After this period, the slave interface will lose arbitrartion for the RAM, and the CPU access can proceed.
COP	5:1	8	CPU Ownership Period. The number of cycles the CPU is guaranteed to own the RAM after it has won the arbitration for the RAM. No arbitration will be performed during this period.
SIE	0	1	Slave Interface Enable. If this bit is set, the slave interface is enabled. Otherwise, the slave interface is disabled and any slave access will be stalled.

ECR - Exception Cause Register

This register identifies the cause of the most recently executed exception. This information may be used to handle exceptions more efficiently in certain operating systems. The register is updated with a value equal to the EVBA offset of the exception, shifted 2 bit positions to the right. Only the 9 lowest bits of the EVBA offset are considered. As an example, an ITLB miss jumps to EVBA+0x50. The ECR will then be loaded with 0x50>>2 == 0x14. The ECR register is not loaded when an *scall*, Breakpoint or OCD Stop CPU exception is taken. Note that for interrupts, the offset is given by the autovector provided by the interrupt controller. The resulting ECR value may therefore overlap with an ECR value used by a regular exception. This can be avoided by choosing the autovector offsets so that no such overlaps occur.

RSR_DBG - Return Status Register for Debug Mode

When Debug mode is entered, the status register contents of the original mode is automatically saved in this register. When the debug routine is finished, the *retd* instruction copies the contents of RSR_DBG into SR.

RAR_DBG - Return Address Register for Debug Mode

When Debug mode is entered, the Program Counter contents of the original mode is automatically saved in this register. When the debug routine is finished, the *retd* instruction copies the contents of RAR_DBG into PC.

CONFIG0 / 1 - Configuration Register 0 / 1

Used to describe the processor, its configuration and capabilities. The contents and functionality of these registers is described in detail in Section 2.7 ”Configuration Registers” on page 17.

COUNT - Cycle Counter Register

Can be used as a general counter to time for example execution time. Can also be used together with COMPARE to implement a periodic interrupt for example for an OS timer. The contents and functionality of this register is described in detail in Section 2.6 ”COMPARE and COUNT registers” on page 17.

COMPARE - Cycle Counter Compare Register

Used together with COUNT to implement a periodic interrupt for example for an OS timer. The contents and functionality of this register is described in detail in Section 2.6 ”COMPARE and COUNT registers” on page 17.

BEAR - Bus Error Address Register

Physical address that caused a Data Bus Error. This register is Read Only. Writes are allowed, but are ignored.

MPUARn - MPU Address Register n

Registers that define the base address and size of the protection regions. Refer to the AVR32 Architecture Manual for details.

MPUPSRn - MPU Privilege Select Register n

Registers that define which privilege register set to use for the different subregions in each protection region. Refer to the AVR32 Architecture Manual for details.

MPUCRA / MPUCRB - MPU Cacheable Register A / B

Registers that define if the different protection regions are cacheable. Refer to the AVR32 Architecture Manual for details.

MPUBRA / MPUBRB - MPU Bufferable Register A / B

Registers that define if the different protection regions are bufferable. Refer to the AVR32 Architecture Manual for details.

MPUAPRA / MPUAPRB - MPU Access Permission Register A / B

Registers that define the access permissions for the different protection regions. Refer to the AVR32 Architecture Manual for details.

MPUCR - MPU Control Register

Register that control the operation of the MPU. Refer to the AVR32 Architecture Manual for details.

SS_STATUS - Secure State Status Register

Register that can be used to pass status or other information from the secure state to the nonsecure state. Refer to Section 5. ”Secure State” on page 59 for details.

SS_ADRF, SS_ADRR, SS_ADR0, SS_ADR1 - Secure State Address Registers

Registers used to partition memories into a secure and a nonsecure section. The 10 LSBs must always be written to zero. Refer to Section 5. ”Secure State” on page 59 for details.

SS_SP_SYS, SS_SP_APP - Secure State SP_SYS and SP_APP Registers

Read-only registers containing the SP_SYS and SP_APP values. Refer to Section 5. ”Secure State” on page 59 for details.

SS_RAR, SS_RSR - Secure State Return Address and Return Status Registers

Contains the address and status register of the *sscall* instruction that called secure state. Also used when returning to nonsecure state with the *retss* instruction. Refer to Section 5. ”Secure State” on page 59 for details.

MSU_ADDRHI, MSU_ADDRLO, MSU_LENGTH, MSU_CTRL, MSU_STATUS, MSU_DATA, MSU_TAIL - Memory Service Unit Registers

These registers are system register mappings of the Memory Service Unit Registers. Refer to Section 9.8 ”Memory Service Unit” on page 138 for details.

2.6 COMPARE and COUNT registers

The COUNT register increments once every clock cycle, regardless of pipeline stalls and flushes. The COUNT register can both be read and written. The COUNT register can be used together with the COMPARE register to create a timer with periodic interrupt. The COUNT register is written to zero upon reset and compare match if the CPUCR[NOCOMPRES] bit is cleared, otherwise COUNT is not reset on compare match. Incrementation of the COUNT register can not be disabled. The COUNT register will increment even though a compare interrupt is pending.

The COMPARE register holds a value that the COUNT register is compared against. The COMPARE register can both be read and written. When the COMPARE and COUNT registers match, a compare interrupt request is generated and COUNT is reset to 0. COUNT will thereafter continue incrementing in the following clock cycle. The interrupt request is routed out to the interrupt controller, which may forward the request back to the processor as a normal interrupt request at a priority level determined by the interrupt controller. Writing a value to the COMPARE register clears any pending compare interrupt requests. The compare and exception generation feature is disabled if the COMPARE register contains the value zero. The COMPARE register is written to zero upon reset.

COUNT and COMPARE are clocked by a dedicated clock with the same frequency as the CPU clock. This allows them to operate in some of the sleep modes. They can therefore be used as timers even when the system use sleep modes. Consult the clock system documentation for information on which sleep modes COUNT and COMPARE are operational.

2.7 Configuration Registers

Configuration registers are used to inform applications and operating systems about the setup and configuration of the processor on which it is running, see Figure 2-4 on page 17.

AVR32UC implements the following read-only configuration registers.

Figure 2-4. Configuration Registers

CONFIG0

31 24	23 20	19 16	15 13	12 10	9 7	6	5	4	3	2	1	0
Processor ID	-	Processor Revision	AT	AR	MMUT	F	J	P	O	S	D	R

CONFIG1

31 26	25 20	19 16	15 13	12 10	9 6	5 3	2 0
IMMU SZ	DMMU SZ	ISET	ILSZ	IASS	DSET	DLSZ	DASS

Table 2-4 on page 18 shows the CONFIG0 fields.

Table 2-4. CONFIG0 Fields

Name	Bit	Description	
Processor ID	31:24	Specifies the type of processor. This allows the application to distinguish between different processor implementations.	
RESERVED	23:20	Reserved for future use.	
Processor revision	19:16	Specifies the revision of the processor implementation.	
AT	15:13	Architecture type	
		Value	Semantic
		0	AVR32A
		1	Unused in AVR32UC
		Other	Reserved
AR	12:10	Architecture Revision	
		Value	Semantic
		0	Unused in AVR32UC
		1	Revision 1
		2	Revision 2
		3	Revision 3
		Other	Reserved
MMUT	9:7	MMU type	
		Value	Semantic
		0	None, using direct mapping and no segmentation
		1	Unused in AVR32UC
		2	Unused in AVR32UC
		3	Memory Protection Unit
		Other	Reserved
F	6	Floating-point unit implemented	
		Value	Semantic
		0	No FPU implemented
		1	Floating-Point Unit implemented
J	5	Java extension implemented	
		Value	Semantic
		0	No Java extension implemented
		1	Unused in AVR32UC
P	4	Performance counters implemented	
		Value	Semantic
		0	No Performance Counters implemented
		1	Unused in AVR32UC

Table 2-4. CONFIG0 Fields (Continued)

Name	Bit	Description	
O	3	On-Chip Debug implemented	
		Value	Semantic
		0	No OCD implemented
		1	OCD implemented
S	2	SIMD instructions implemented	
		Value	Semantic
		0	No SIMD instructions
		1	Unused in AVR32UC
D	1	DSP instructions implemented	
		Value	Semantic
		0	Unused in AVR32UC
		1	DSP instructions implemented
R	0	Memory Read-Modify-Write instructions implemented	
		Value	Semantic
		0	Unused in AVR32UC
		1	RMW instructions implemented

Table 2-5 on page 19 shows the CONFIG1 fields.

Table 2-5. CONFIG1 Fields

Name	Bit	Description
IMMU SZ	31:26	Unused in AVR32UC
DMMU SZ	25:20	Specifies the number of MPU entries.
ISET	19:16	Unused in AVR32UC
ILSZ	15:13	Unused in AVR32UC
IASS	12:10	Unused in AVR32UC
DSET	9:6	Unused in AVR32UC
DLSZ	5:3	Unused in AVR32UC
DASS	2:0	Unused in AVR32UC

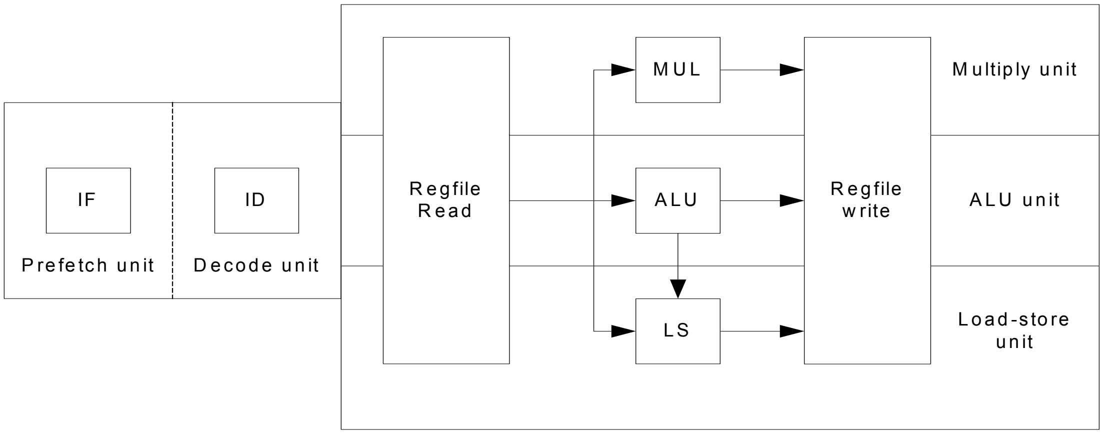
3. Pipeline

3.1 Overview

AVR32UC is a pipelined processor with three pipeline stages: IF, ID and EX. All instructions are issued and complete in order. Some instructions may require several iterations through the EX stage in order to complete.

The following figure shows an overview of the AVR32UC pipeline stages.

Figure 3-1. The AVR32UC pipeline stages.



The follwing abbreviations are used in the figure:

- IF - Instruction Fetch
- ID - Instruction Decode
- EX - Instruction Execute
- MUL - Multiplier
- ALU - Arithmetic-Logic Unit
- LS - Load/Store Unit

3.2 Prefetch unit

The prefetch unit comprises the IF pipestage, and is responsible for feeding instructions to the decode unit. The prefetch unit fetches 32 bits at a time from the instruction memory interface and places them in a FIFO prefetch buffer. At the same time, one instruction, either RISC extended or compact, is fed to the decode stage.

3.3 Decode unit

The decode unit generates the necessary signals in order for the instruction to execute correctly. The ID stage accepts one instruction each clock cycle from the prefetch unit. This instruction is then decoded, and control signals and register file addresses are generated. If the instruction cannot be decoded, an illegal instruction or unimplemented instruction exception is issued. The ID stage also contains a state machine required for controlling multicycle instructions.

The ID stage performs the remapping of register file addresses from logical to physical addresses. This is used for remapping the stack pointer register into the SP_APP or SP_SYS registers.

3.4 EX pipeline stage

The Execute (EX) pipeline stage performs register file reads, operations on registers and memory, and register file writes.

3.4.1 ALU section

The ALU pipeline performs most of the data manipulation instructions, like arithmetical and logical operations. The ALU stage performs the following tasks:

- Target address calculation and condition check for change-of-flow instructions.
- Condition code checking for conditional instructions.
- Address calculation for memory accesses
- Writeback address calculation for the LS pipeline.
- All flag setting for arithmetical and logical instructions.
- The saturation needed by *satadd* and *satsub*.
- The operation needed by *satrnds*, *satrndu*, *sats* and *satu*.
- Signed and unsigned division

3.4.2 Multiply section

All multiply instructions execute in the multiply section. This section implements a 32 by 32 multiplier array, and 16x16, 32x16 and 32x32 multiplications and multiply-accumulates therefore have an issue latency of one cycle. Multiplication of 32 by 32 bits to a 64-bit result require two iterations through the multiplier array, and therefore needs several cycles to complete. This will stall the multiply pipeline until the instruction is complete.

A special accumulator cache is implemented in the MUL section. This cache saves the multiply-accumulate result in dedicated registers in the MUL section, as well as writing them back to the register file. This allows subsequent MAC instructions to read the accumulator value from the cache, instead of from the register file. This will speed up MAC operations by one clock cycle. If a MAC instruction targets a register not found in the cache, one clock cycle is added to the MAC operation, loading the accumulator value from the register file into the cache. In the next cycle, the MAC operation is restarted automatically by hardware. If an instruction, like an add, mul or load, is executed with target address equal to that of a valid cached register, the instruction will update the cache.

The accumulator cache can hold one doubleword accumulator value, or one word accumulator value. Hardware ensures that the accumulator cache is kept consistent. If another pipeline section writes to one of the registers kept in the accumulator cache, the cache is updated. The cache is automatically invalidated after reset.

3.4.3 Load-store section

The load-store (LS) pipeline is able to read or write one register per clock cycle. The address is calculated by the ALU section. Thereafter the address is passed on to the LS section and output to the memory interface, together with the data to write if the access is a write. If the access is a read, the read data is returned from the memory interface in the same cycle. If the read data requires typecasting or other manipulation like performed by *ldins* or *ldswp*, this manipulation is performed in the same cycle.

Any load or store multiple registers are decoded by the ID stage and passed on to the EX stage as a series of single load or store word operations.

The read-modify-write instructions *memc*, *mems* and *memt* are performed as a non-interruptable sequence of read from and write to memory. The load-store section generates the control signals required to perform this sequence. This sequence takes several clock cycles, so any following instructions requiring the use of the load-store section must stall until the sequence is finished. Following instructions that do not use the load-store section will not have to stall even if the sequence has not finished.

Some memory operations to slow memories, such as memories on the HSB bus, may require several clock cycles to perform. If required, the CPU pipeline will stall as long as necessary in order to perform the memory access.

3.5 Support for unaligned addresses

All memory accesses must be performed with the correct alignment according to the data size. The only exception to this is doubleword accesses, which are performed as two word accesses, and therefore can be word-aligned. Any other unaligned memory access will cause an Data Address Exception.

Instruction fetches must be halfword aligned. Any other alignment will cause an Instruction Address Exception.

3.6 Forwarding hardware and hazard detection

Since the register file is read and written in the same pipeline stage, no hazards can occur, and no forwarding is necessary. The programmer does not need to take any special considerations regarding data hazards when writing code.

3.7 Event handling

Due to various reasons, the CPU may be required to abort normal program execution in order to handle special, high-priority events. When handling of these events is complete, normal program execution can be resumed. Traditionally, events that are generated internally in the CPU are called exceptions, while events generated by sources external to the CPU are called interrupts. The possible sources of events are listed in Table 3-4 on page 28.

The AVR32 has a powerful event handling scheme. The different event sources, like Illegal Opcode and external interrupt requests, have different priority levels, ensuring a well-defined behaviour when multiple events are received simultaneously. Additionally, pending events of a higher priority class may preempt handling of ongoing events of a lower priority class.

When an event occurs, the execution of the instruction stream is halted, and execution control is passed to an event handler at an address specified in Table 3-4 on page 28. Most of the handlers are placed sequentially in the code space starting at the address specified by EVBA, with four bytes between each handler. This gives ample space for a jump instruction to be placed there, jumping to the event routine itself. A few critical handlers have larger spacing between them, allowing the entire event routine to be placed directly at the address specified by the EVBA-relative offset generated by hardware. All external interrupt sources have autovectored interrupt service routine (ISR) addresses. This allows the interrupt controller to directly specify the ISR address as an address relative to EVBA. The autovector offset has 14 address bits, giving an offset of maximum 16384 bytes. The target address of the event handler is calculated as (EVBA | event_handler_offset), not (EVBA + event_handler_offset), so EVBA and exception code segments must be set up appropriately.

The same mechanisms are used to service all different types of events, including external interrupt requests, yielding a uniform event handling scheme.

Each pipeline stage has a pipeline register that holds the exception requests associated with the instruction in that pipeline stage. This allows the exception request to follow the contaminated instruction through the pipeline. Exceptions are detected in two different pipeline stages. The EX stage detects all data-address related exceptions (DTLB Protection and Data Address). All other exceptions, including interrupts, are detected in the ID stage. When an exception is detected in EX, the EX stage and all upstream stages are flushed.

Generally, all exceptions, including breakpoint, have the failing instruction as restart address. This allows a fixup exception routine to correct the error and restart the instruction. Interrupts (INT0-3, NMI) have the address of the first non-completed instruction as restart address.

3.7.1 Exceptions and interrupt requests

When an event other than *scall* or debug request is received by the core, the following actions are performed atomically:

1. The pending event will not be accepted if it is masked. The I3M, I2M, I1M, I0M, EM and GM bits in the Status Register are used to mask different events. Not all events can be masked. A few critical events (NMI, Unrecoverable Exception, TLB Multiple Hit and Bus Error) can not be masked. When an event is accepted, hardware automatically sets the mask bits corresponding to all sources with equal or lower priority. This inhibits acceptance of other events of the same or lower priority, except for the critical events listed above. Software may choose to clear some or all of these bits after saving the necessary state if other priority schemes are desired. It is the event source's responsibility to ensure that their events are left pending until accepted by the CPU.
2. When a request is accepted, the Status Register and Program Counter of the current context is stored to the system stack. If the event is an INT0, INT1, INT2 or INT3, registers R8-R12 and LR are also automatically stored to stack. Storing the Status Register ensures that the core is returned to the previous execution mode when the current event handling is completed. When exceptions occur, both the EM and GM bits are set, and the application may manually enable nested exceptions if desired by clearing the appropriate bit. Each exception handler has a dedicated handler address, and this address uniquely identifies the exception source.
3. The Mode bits are set to reflect the priority of the accepted event, and the correct register file bank is selected. The address of the event handler, as shown in Table 3-4, is loaded into the Program Counter.

The execution of the event handler routine then continues from the effective address calculated.

The *rete* instruction signals the end of the event. When encountered, the Return Status Register and Return Address Register are popped from the system stack and restored to the Status Register and Program Counter. If the *rete* instruction returns from INT0, INT1, INT2 or INT3, registers R8-R12 and LR are also popped from the system stack. The restored Status Register contains information allowing the core to resume operation in the previous execution mode. This concludes the event handling.

Note that event priorities are only used to determine which event handler to call first when multiple events are received simultaneously. Once control is passed on to the event handler, handling of pending and lower priority events may be initiated if not masked.

For instance, it is possible to make a supervisor call (SCALL) from an interrupt level 0 handler, even though the priority of a supervisor call event is lower than the active interrupt level 0 event.

3.7.2 Supervisor calls

The AVR32 instruction set provides a supervisor mode call instruction. The *scall* instruction is designed so that privileged routines can be called from any context. This facilitates sharing of code between different execution modes. The *scall* mechanism is designed so that a minimal execution cycle overhead is experienced when performing supervisor routine calls from time-critical event handlers.

The *scall* instruction behaves differently depending on which mode it is called from. The behaviour is detailed in the instruction set reference. In order to allow the *scall* routine to return to the correct context, a return from supervisor call instruction, *rets*, is implemented. In the AVR32A microarchitecture, *scall* and *rets* uses the system stack to store the return address and the status register.

3.7.3 Debug requests

The AVR32 architecture defines a dedicated debug mode. When a debug request is received by the core, Debug mode is entered. Entry into Debug mode can be masked by the DM bit in the status register. Upon entry into Debug mode, hardware sets the SR[D] bit and jumps to the Debug Exception handler. By default, debug mode executes in the exception context, but with dedicated Return Address Register and Return Status Register. These dedicated registers remove the need for storing this data to the system stack, thereby improving debuggability.

Debug mode is exited by executing the *retd* instruction. This returns to the previous context.

3.8 Special concerns

3.8.1 System stack

Event handling in AVR32UC, like in all AVR32A architectures, uses the system stack pointed to by the system stack pointer, SP_SYS, for pushing and popping R8-R12, LR, status register and return address. Since exception code may be timing-critical, SP_SYS should point to memory addresses in the IRAM section, since the timing of accesses to this memory section is both fast and deterministic.

The user must also make sure that the system stack is large enough so that any event is able to push the required registers to stack. If the system stack is full, and an event occurs, the system will enter an UNDEFINED state.

3.8.2 Clearing of pending interrupt requests

When an interrupt request is accepted by the CPU, the interrupt handler will eventually be called. The interrupt handler is responsible for performing the required actions so that the requesting module disasserts the interrupt request before the interrupt routine is exited with *rete*. Failing to do so will cause the interrupt handler to be re-entered after the *rete* instruction has been executed, since the interrupt request is still active. Different interrupt sources have different ways of disasserting requests, for example reading an interrupt cause register or writing to specific control registers. Refer to the module-specific documentation for information on how to disassert interrupt requests.

Disasserting an interrupt request often requires that a bus access is performed to the requesting module. An example of such an access is to read an interrupt cause register. There will be a latency from the execution of the load or store instruction that is to disassert the interrupt request and the actual disassertion of the request. This latency can be caused by the bus system and internal latencies in the interrupting module. It is important that the programmer makes sure that the interrupt request has actually been disasserted before returning from the interrupt with *rete*.

This can usually be ensured by scheduling the code sequence disasserting the interrupt request in such a way that one can be certain that the interrupt request has actually been disasserted before the *rete* instruction is executed.

Code 3-1. Clearing IRQs using code scheduling

```
// Using scheduling of instructions in the IRQ handler to make sure that the
// request has been disasserted before returning from the handler.
// Assume that the IRQ is cleared by reading PERIPH_INTCAUSE, r0 points to
// this register.
irq_handler:
    <some instructions>
    ld.w r12, r0[0] // Clear the IRQ
    <some other instructions, enough to make sure that the IRQ is cleared>
    rete
```

The mechanisms and timing required for disasserting an interrupt request from a module is specific to different modules. Usually, the request is disasserted within a few clock cycles after the load or store instruction has been received by the module. In this case, a simple way of making sure that the request has actually been disasserted is to use a data memory barrier (“Data memory barriers” on page 64). The DMB will block the CPU pipeline until the interrupt request has been disasserted. At this point, the *rete* instruction can safely be executed.

Code 3-2. Clearing IRQs using data memory barriers

```
// Using data memory barriers in the IRQ handler to make sure that the
// request has been disasserted before returning from the handler
// Assume that the IRQ is cleared by writing a bitmask to PERIPH_INTCLEAR.
// r0 points to this register, r1 contains the correct bitmask.
irq_handler:
    <some instructions>
    st.w r0[0], r1
    ld.w r12, r0[0] // data memory barrier
    rete
```

The programmer should consult the data sheets for the different peripheral modules to check if special timings or concerns related to disasserting of interrupt requests apply to the specific module.

3.8.3 Masking interrupt requests in peripheral modules

Handling an interrupt request involves several operations like pushing of registers to stack and takes several clock cycles. The required operations are controlled by sequencing logic in hardware. This sequencing hardware does not permit that an asserted interrupt request is disasserted while it is in the process of handling this interrupt request.

Hardware makes sure that manipulation of the GM and IxM bits in SREG can be performed safely at all times using the *mtsr*, *csrf* and *ssrf* instructions. The programmer does not need to take any special concerns when issuing one of these instructions.

All hardware connected to the CPU is implemented in such a way that once an interrupt request is asserted by the hardware, it can only be disasserted by explicit actions by the programmer.

Many peripheral modules that are able to assert interrupt requests have control registers or other means of masking one or more of its interrupt requests. For example, a USART can contain an interrupt mask register with individual bits for masking "TX ready" and "RX ready" interrupts. Writing to such a mask register may cause a pending interrupt request from that module to be disasserted.

The programmer must at all times make sure that an action that will disassert interrupts at the interrupt source is not performed if it is possible that the interrupt sequencing hardware is in the processing of handling the interrupt request that will be disasserted by the action. It is safe to perform such an action if one of the following is true:

- The SREG GM or IxM bit corresponding to the priority of the interrupt request to be masked is set before the action is performed.
- It can be guaranteed that the interrupt request being masked by the action is disasserted when the action is initiated and being performed.

Code 3-3. Masking IRQs in a peripheral module which may assert an IRQ at any time

```
// Masking TX_READY IRQ in a peripheral by setting the TXMASK bit in the
// IRQMASK register of the peripheral.
// Could alternatively mask the SREG IxM bit associated with the IRQ source
disassert_periph_tx_irq:
    ssrf AVR32_SREG_GM
    mems PERIPH_IRQMASK, PERIPH_TXMASK
    csrf AVR32_SREG_GM
```

If the interrupt request is disasserted during the critical clock cycles where the sequencing hardware is active handling this interrupt request, the CPU may enter an UNPREDICTABLE state.

3.9 Entry points for events

Several different event handler entry points exists. In AVR32UC, the reset address is 0x8000_0000. This places the reset address in the boot flash memory area.

TLB miss exceptions and *scall* have a dedicated space relative to EVBA where their event handler can be placed. This speeds up execution by removing the need for a jump instruction placed at the program address jumped to by the event hardware. All other exceptions have a dedicated event routine entry point located relative to EVBA. The handler routine address identifies the exception source directly.

AVR32UC uses the ITLB and DTLB protection exceptions to signal a MPU protection violation. ITLB and DTLB miss exceptions are used to signal that an access address did not map to any of the entries in the MPU. TLB multiple hit exception indicates that an access address did map to multiple TLB entries, signalling an error.

All external interrupt requests have entry points located at an offset relative to EVBA. This autovector offset is specified by an external Interrupt Controller. The programmer must make sure that none of the autovector offsets interfere with the placement of other code. The autovector offset has 14 address bits, giving an offset of maximum 16384 bytes.

Special considerations should be made when loading EVBA with a pointer. Due to security considerations, the event handlers should be located in non-writeable flash memory, or optionally in a privileged memory protection region if an MPU is present.

If several events occur on the same instruction, they are handled in a prioritized way. The priority ordering is presented in Table 3-4. If events occur on several instructions at different locations in the pipeline, the events on the oldest instruction are always handled before any events on any younger instruction, even if the younger instruction has events of higher priority than the oldest instruction. An instruction B is younger than an instruction A if it was sent down the pipeline later than A.

The addresses and priority of simultaneous events are shown in Table 3-4 on page 28. Some of the exceptions are unused in AVR32UC since it has no MMU or coprocessor interface.

The interrupt system requires that an interrupt controller is present outside the core in order to prioritize requests and generate a correct offset if more than one interrupt source exists for each priority level. An interrupt controller generating different offsets depending on interrupt request source is referred to as autovectoring. Note that the interrupt controller should generate autovector addresses that do not conflict with addresses in use by other events or regular program code.

The addresses of the interrupt routines are calculated by adding the address on the autovector offset bus to the value of the Exception Vector Base Address (EVBA). The INT0, INT1, INT2, INT3, and NMI signals indicate the priority of the pending interrupt. INT0 has the lowest priority, and NMI the highest priority of the interrupts.

Table 3-4. Priority and handler addresses for events

Priority	Handler Address	Name	Event source	Stored Return Address
1	0x8000_0000	Reset	External input	Undefined
2	Provided by OCD system	OCD Stop CPU	OCD system	First non-completed instruction
3	EVBA+0x00	Unrecoverable exception	Internal	PC of offending instruction
4	EVBA+0x04	TLB multiple hit	MPU	PC of offending instruction
5	EVBA+0x08	Bus error data fetch	Data bus	First non-completed instruction
6	EVBA+0x0C	Bus error instruction fetch	Data bus	First non-completed instruction
7	EVBA+0x10	NMI	External input	First non-completed instruction
8	Autovectored	Interrupt 3 request	External input	First non-completed instruction
9	Autovectored	Interrupt 2 request	External input	First non-completed instruction
10	Autovectored	Interrupt 1 request	External input	First non-completed instruction
11	Autovectored	Interrupt 0 request	External input	First non-completed instruction
12	EVBA+0x14	Instruction Address	CPU	PC of offending instruction
13	EVBA+0x50	ITLB Miss	MPU	PC of offending instruction
14	EVBA+0x18	ITLB Protection	MPU	PC of offending instruction
15	EVBA+0x1C	Breakpoint	OCD system	First non-completed instruction
16	EVBA+0x20	Illegal Opcode	Instruction	PC of offending instruction
17	EVBA+0x24	Unimplemented instruction	Instruction	PC of offending instruction
18	EVBA+0x28	Privilege violation	Instruction	PC of offending instruction
19	EVBA+0x2C	Floating-point	UNUSED	
20	EVBA+0x30	Coprocessor absent	Coprocessor	PC of offending instruction
21	EVBA+0x100	Supervisor call	Instruction	PC(Supervisor Call) +2
22	EVBA+0x34	Data Address (Read)	CPU	PC of offending instruction
23	EVBA+0x38	Data Address (Write)	CPU	PC of offending instruction
24	EVBA+0x60	DTLB Miss (Read)	MPU	PC of offending instruction
25	EVBA+0x70	DTLB Miss (Write)	MPU	PC of offending instruction
26	EVBA+0x3C	DTLB Protection (Read)	MPU	PC of offending instruction
27	EVBA+0x40	DTLB Protection (Write)	MPU	PC of offending instruction
28	EVBA+0x44	DTLB Modified	UNUSED	

3.9.1 Description of events

3.9.1.1 Reset Exception

The Reset exception is generated when the reset input line to the CPU is asserted. The Reset exception can not be masked by any bit. The Reset exception resets all synchronous elements and registers in the CPU pipeline to their default value, and starts execution of instructions at address 0x8000_0000.

```
SR = reset_value_of_SREG;
```

```
PC = 0x8000_0000;
```

All other system registers are reset to their reset value, which may or may not be defined. Refer to the Programming Model chapter for details.

3.9.1.2 *OCD Stop CPU Exception*

The OCD Stop CPU exception is generated when the OCD Stop CPU input line to the CPU is asserted. The OCD Stop CPU exception can not be masked by any bit. This exception is identical to a non-maskable, high priority breakpoint. Any subsequent operation is controlled by the OCD hardware. The OCD hardware will take control over the CPU and start to feed instructions directly into the pipeline.

```
RSR_DBG = SR;
RAR_DBG = PC;
SR[M2:M0] = B’110;
SR[D] = 1;
SR[DM] = 1;
SR[EM] = 1;
SR[GM] = 1;
```

3.9.1.3 *Unrecoverable Exception*

The Unrecoverable Exception is generated when an exception request is issued when the Exception Mask (EM) bit in the status register is asserted. The Unrecoverable Exception can not be masked by any bit. The Unrecoverable Exception is generated when a condition has occurred that the hardware cannot handle. The system will in most cases have to be restarted if this condition occurs.

```
*(--SP_SYS) = PC of offending instruction;
*(--SP_SYS) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x00;
```

3.9.1.4 *TLB Multiple Hit Exception*

The TLB Multiple Hit Exception is generated when an access hits in multiple MPU regions. This is usually caused by programming error. Used only if an MPU is present.

```
*(--SP_SYS) = PC of offending instruction;
*(--SP_SYS) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x04;
```

3.9.1.5 *Bus Error Exception on Data Access*

The Bus Error on Data Access exception is generated when the data bus detects an error condition. This exception is caused by events unrelated to the instruction stream, or by data written to the cache write-buffers many cycles ago. Therefore, execution can not be resumed in a safe way after this exception. The return address placed on stack is unrelated to the operation that caused the exception. The exception handler is responsible for performing the appropriate action.

```
*(--SP$_{SYS}$) = PC of first non-issued instruction;
*(--SP$_{SYS}$) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x08;
BEAR = failing address
```

3.9.1.6 *Bus Error Exception on Instruction Fetch*

The Bus Error on Instruction Fetch exception is generated when the data bus detects an error condition. This exception is caused by events related to the instruction stream. Therefore, execution can be restarted in a safe way after this exception, assuming that the condition that caused the bus error is dealt with.

```
*(--SP$_{SYS}$) = PC of first non-issued instruction;
*(--SP$_{SYS}$) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x0C;
```

3.9.1.7 *NMI Exception*

The NMI exception is generated when the NMI input line to the core is asserted. The NMI exception can not be masked by the SR[GM] bit. However, the core ignores the NMI input line when processing an NMI Exception (the SR[M2:M0] bits are B’111). This guarantees serial execution of NMI Exceptions, and simplifies the NMI hardware and software mechanisms.

Since the NMI exception is unrelated to the instruction stream, the instructions in the pipeline are allowed to complete. After finishing the NMI exception routine, execution should continue at the instruction following the last completed instruction in the instruction stream.

```
*(--SP$_{SYS}$) = PC of first noncompleted instruction;
*(--SP$_{SYS}$) = SR;
SR[M2:M0] = B’111;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x10;
```

3.9.1.8 *INT3 Exception*

The INT3 exception is generated when the INT3 input line to the core is asserted. The INT3 exception can be masked by the SR[GM] bit, and the SR[I3M] bit. Hardware automatically sets the SR[I3M] bit when accepting an INT3 exception, inhibiting new INT3 requests when processing an INT3 request.

The INT3 Exception handler address is calculated by adding EVBA to an interrupt vector offset specified by an interrupt controller outside the core. The interrupt controller is responsible for providing the correct offset.

Since the INT3 exception is unrelated to the instruction stream, the instructions in the pipeline are allowed to complete. After finishing the INT3 exception routine, execution should continue at the instruction following the last completed instruction in the instruction stream.

```
*(--$SP_{SYS}$) = R8;
*(--$SP_{SYS}$) = R9;
*(--$SP_{SYS}$) = R10;
*(--$SP_{SYS}$) = R11;
*(--$SP_{SYS}$) = R12;
*(--$SP_{SYS}$) = LR;
*(--$SP_{SYS}$) = PC of first noncompleted instruction;
*(--$SP_{SYS}$) = SR;
SR[M2:M0] = B’101;
SR[I3M] = 1;
SR[I2M] = 1;
SR[I1M] = 1;
SR[I0M] = 1;
PC = EVBA | INTERRUPT_VECTOR_OFFSET;
```

3.9.1.9 *INT2 Exception*

The INT2 exception is generated when the INT2 input line to the core is asserted. The INT2 exception can be masked by the SR[GM] bit, and the SR[I2M] bit. Hardware automatically sets the SR[I2M] bit when accepting an INT2 exception, inhibiting new INT2 requests when processing an INT2 request.

The INT2 Exception handler address is calculated by adding EVBA to an interrupt vector offset specified by an interrupt controller outside the core. The interrupt controller is responsible for providing the correct offset.

Since the INT2 exception is unrelated to the instruction stream, the instructions in the pipeline are allowed to complete. After finishing the INT2 exception routine, execution should continue at the instruction following the last completed instruction in the instruction stream.

```
*(--$SP_{SYS}$) = R8;
*(--$SP_{SYS}$) = R9;
*(--$SP_{SYS}$) = R10;
*(--$SP_{SYS}$) = R11;
*(--$SP_{SYS}$) = R12;
*(--$SP_{SYS}$) = LR;
*(--$SP_{SYS}$) = PC of first noncompleted instruction;
*(--$SP_{SYS}$) = SR;
SR[M2:M0] = B’100;
SR[I2M] = 1;
SR[I1M] = 1;
SR[I0M] = 1;
PC = EVBA | INTERRUPT_VECTOR_OFFSET;
```

3.9.1.10 *INT1 Exception*

The INT1 exception is generated when the INT1 input line to the core is asserted. The INT1 exception can be masked by the SR[GM] bit, and the SR[I1M] bit. Hardware automatically sets the SR[I1M] bit when accepting an INT1 exception, inhibiting new INT1 requests when processing an INT1 request.

The INT1 Exception handler address is calculated by adding EVBA to an interrupt vector offset specified by an interrupt controller outside the core. The interrupt controller is responsible for providing the correct offset.

Since the INT1 exception is unrelated to the instruction stream, the instructions in the pipeline are allowed to complete. After finishing the INT1 exception routine, execution should continue at the instruction following the last completed instruction in the instruction stream.

```
*(--$SP_{SYS}$) = R8;
*(--$SP_{SYS}$) = R9;
*(--$SP_{SYS}$) = R10;
*(--$SP_{SYS}$) = R11;
*(--$SP_{SYS}$) = R12;
*(--$SP_{SYS}$) = LR;
*(--$SP_{SYS}$) = PC of first noncompleted instruction;
*(--$SP_{SYS}$) = SR;
SR[M2:M0] = B’011;
SR[I1M] = 1;
SR[I0M] = 1;
PC = EVBA | INTERRUPT_VECTOR_OFFSET;
```

3.9.1.11 INT0 Exception

The INT0 exception is generated when the INT0 input line to the core is asserted. The INT0 exception can be masked by the SR[GM] bit, and the SR[I0M] bit. Hardware automatically sets the SR[I0M] bit when accepting an INT0 exception, inhibiting new INT0 requests when processing an INT0 request.

The INT0 Exception handler address is calculated by adding EVBA to an interrupt vector offset specified by an interrupt controller outside the core. The interrupt controller is responsible for providing the correct offset.

Since the INT0 exception is unrelated to the instruction stream, the instructions in the pipeline are allowed to complete. After finishing the INT0 exception routine, execution should continue at the instruction following the last completed instruction in the instruction stream.

```
*(--$SP_{SYS}$) = R8;
*(--$SP_{SYS}$) = R9;
*(--$SP_{SYS}$) = R10;
*(--$SP_{SYS}$) = R11;
*(--$SP_{SYS}$) = R12;
*(--$SP_{SYS}$) = LR;
*(--$SP_{SYS}$) = PC of first noncompleted instruction;
*(--$SP_{SYS}$) = SR;
SR[M2:M0] = B’010;
SR[I0M] = 1;
PC = EVBA | INTERRUPT_VECTOR_OFFSET;
```

3.9.1.12 Instruction Address Exception

The Instruction Address Error exception is generated if the generated instruction memory address has an illegal alignment.

```
*(--SP_SYS) = PC;
*(--SP_SYS) = SR;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x14;
```

3.9.1.13 ITLB Miss Exception

The ITLB Miss exception is generated when the MPU is enabled and the instruction memory access does not hit in any regions. Used only if an MPU is present.

```
*(--SP_SYS) = PC;
*(--SP_SYS) = SR;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x50;
```

3.9.1.14 ITLB Protection Exception

The ITLB Protection exception is generated when the instruction memory access violates the access rights specified by the protection region in which the address lies. Used only if an MPU is present.

```
*(--SP_SYS) = PC;
*(--SP_SYS) = SR;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x18;
```

3.9.1.15 Breakpoint Exception

The Breakpoint exception is issued when the OCD breakpoint input line to the CPU is aserted, and SREG[DM] is cleared.

When entering the exception routine, RAR_DBG points to the *breakpoint* instruction, and the CPU will enter Debug mode. An external debugger can optionally assume control of the CPU when the Breakpoint Exception is executed. The debugger can then issue individual instructions to be executed in Debug mode. Debug mode is exited with the *retd* instruction. This passes control from the debugger back to the CPU, resuming normal execution.

```
RSR_DBG = SR;
RAR_DBG = PC;
SR[M2:M0] = B'110;
SR[D] = 1;
SR[DM] = 1;
SR[EM] = 1;
SR[GM] = 1;
```

```
PC = EVBA | 0x1C;
```

3.9.1.16 *Illegal Opcode*

This exception is issued when the core fetches an unknown instruction, or when a coprocessor instruction is not acknowledged. When entering the exception routine, the return address on stack points to the instruction that caused the exception.

```
*(--SP_SYS) = PC;
*(--SP_SYS) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x20;
```

3.9.1.17 *Unimplemented Instruction*

This exception is issued when the core fetches an instruction supported by the instruction set but not by the current implementation. This allows software implementations of unimplemented instructions. When entering the exception routine, the return address on stack points to the instruction that caused the exception.

Table 3-5. List of unimplemented instructions.

Privileged Instructions	**Comment**
All SIMD instructions	No SIMD implemented
Coprocessor instructions adressing unimplemented coprocessors	
cache - perform cache operation	No cache implemented
incjosp - increment Java stack pointer	No Java implemented
popjc - pop Java context	No Java implemented
pushjc - push Java context	No Java implemented
retj- return from Java mode	No Java implemented
tlbr - read addressed TLB entry into TLBEHI and TLBELO	No MMU present
tlbw - write TLB entry registers into TLB	No MMU present
tlbs - search TLB for entry matching TLBEHI[VPN]	No MMU present

```
*(--SP_SYS) = PC;
*(--SP_SYS) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x24;
```

3.9.1.18 *Data Read Address Exception*

The Data Read Address Error exception is generated if the address of a data memory read has an illegal alignment.

```
*(--$SP_{SYS}$) = PC;
*(--$SP_{SYS}$) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x34;
```

3.9.1.19 *Data Write Address Exception*

The Data Write Address Error exception is generated if the address of a data memory write has an illegal alignment.

```
*(--$SP_{SYS}$) = PC;
*(--$SP_{SYS}$) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x38;
```

3.9.1.20 *DTLB Read Miss Exception*

The DTLB Read Miss exception is generated when the MPU is enabled and the data memory read access does not hit in any regions. Used only if an MPU is present.

```
*(--$SP_{SYS}$) = PC;
*(--$SP_{SYS}$) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x60;
```

3.9.1.21 *DTLB Write Miss Exception*

The DTLB Write Miss exception is generated when the MPU is enabled and the data memory write access does not hit in any regions. Used only if an MPU is present.

```
*(--$SP_{SYS}$) = PC;
*(--$SP_{SYS}$) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x70;
```

3.9.1.22 *DTLB Read Protection Exception*

The DTLB Protection exception is generated when the data memory read violates the access rights specified by the protection region in which the address lies. Used only if an MPU is present.

```
*(--$SP_{SYS}$) = PC;
*(--$SP_{SYS}$) = SR;
```

```
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x3C;
```

3.9.1.23 DTLB Write Protection Exception

The DTLB Protection exception is generated when the data memory write violates the access rights specified by the protection region in which the address lies. Used only if an MPU is present.

```
*(--SP_SYS) = PC;
*(--SP_SYS) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x40;
```

3.9.1.24 Privilege Violation Exception

If the application tries to execute privileged instructions, this exception is issued. The complete list of priveleged instructions is shown in Table 3-6 on page 36. When entering the exception routine, the address of the instruction that caused the exception is stacked as the return address.

```
*(--SP_SYS) = PC;
*(--SP_SYS) = SR;
SR[M2:M0] = B’110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x28;
```

Table 3-6. List of instructions which can only execute in privileged modes.

Privileged Instructions	Comment
csrf - clear status register flag	Privileged only when accessing upper half of status register
mtsr - move to system register	
mfsr - move from system register	
mtdr - move to debug register	
mfdr - move from debug register	
rete- return from exception	
rets - return from supervisor call	
retd - return from debug mode	
sleep - sleep	
ssrf - set status register flag	Privileged only when accessing upper half of status register

3.9.1.25 DTLB Modified Exception

Unused in AVR32UC, since it has no MMU.

3.9.1.26 *Floating-point Exception*

Unused in AVR32UC.

3.9.1.27 *Coprocessor Absent Exception*

The Coprocessor Absent exception is generated when a nonexisting coprocessor is addressed by a coprocessor instruction. Used only if one or more coprocessors are present. Executing coprocessor instructions in systems with no coprocessors results in an Unimplemented Instruction exception instead.

```
*(--SP_SYS) = PC;
*(--SP_SYS) = SR;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x30;
```

3.9.1.28 *Supervisor call*

Supervisor calls are signalled by the application code executing a supervisor call (*scall*) instruction. The *scall* instruction behaves differently depending on which context it is called from. This allows *scall* to be called from other contexts than Application.

When the exception routine is finished, execution continues at the instruction following *scall*. The *rets* instruction is used to return from supervisor calls.

```
If ( SR[M2:M0] == {B'000 or B'001} )
  *(--SP_SYS) = PC;
  *(--SP_SYS) = SR;
  PC ← EVBA | 0x100;
  SR[M2:M0] ← B'001;
else
  LR ← PC + 2;
  PC ← EVBA | 0x100;
```

3.10 Interrupt latencies

The following features in AVR32UC ensure low and deterministic interrupt latency:

- Four different interrupt levels and an NMI ensures that the user can efficiently prioritize the interrupt sources.
- Long-running instructions such as *ldm*, *stm*, *pushm*, *popm*, *divs* and *divu* will be aborted if an interrupt request is received. The slowest instruction that can not be aborted by a pending interrupt has a worst case issue latency of 5 cycles. This implies that an interrupt request will need to wait at most 5 cycles for an instruction to complete. The fastest instructions need only a single cycle to complete.
- Interrupts are autovectored, allowing the CPU to jump directly to the interrupt handler.
- When an interrupt of level m is received, the CPU will start stacking register file registers, return address and status register. After this stacking is performed, the CPU will jump to the autovector address of the interrupt of level m. If an interrupt of level n, where n > m, is received during this stacking, the CPU will jump to the autovector address of the interrupt of level n, NOT the autovector address of the original interrupt.

Note that the overall system latency from an interrupt request is signaled to the request is being handled depends on a number of things in addition to the latency through the CPU. The latency through the interrupt controller will affect interrupt latency for all peripheral interrupt requests and the bus matrix, code and data memories will affect overall responsiveness.

3.10.1 Maximum interrupt latency

The maximum CPU interrupt latency can be calculated as follows:

Table 3-7. Maximum interrupt latency

Source	Delay
Wait for the slowest instruction to complete	6
Stack register file registers, return address and status register, and jump to autovector target	10
Wait for autovector target instruction to be fetched	1
TOTAL	17

3.10.2 Minimum interrupt latency

The minimum CPU interrupt latency of an interrupt request of level m will occur when the CPU is in the process of stacking the registers and return address associated with an interrupt request of level n, where n < m. If the level m interrupt request arrives just as the CPU is about to jump to the autovector address for the interrupt of level n, the CPU will jump directly to the autovector address of the latest arriving interrupt. In this case, the minimum interrupt latency is as follows:

Table 3-8. Minimum interrupt latency - higher priority interrupt preempts lower priority interrupt

Source	Delay
Jump to autovector target	1
Wait for autovector target instruction to be fetched	1
TOTAL	2

Assuming that the interrupt request arrives when the CPU is in the process of executing program code, the minimum interrupt latency can be calculated as follows:

Table 3-9. Minimum interrupt latency - interrupt received when executing program code

Source	Delay
Wait for the fastest instruction to complete	1
Stack register file registers, return address and status register, and jump to autovector target	10
Wait for autovector target instruction to be fetched	1
TOTAL	12

3.11 NMI latency

Non-maskable interrupts (NMI) behave similarly to interrupts, except that they do not automatically push register file registers on the stack. NMI can, similar to interrupts, abort long-running instructions.

The maximum NMI latency can be calculated as follows:

Table 3-10. Maximum NMI latency

Source	Delay
Wait for the slowest instruction to complete	6
Stack return address and status register, and jump to autovector target	4
Wait for autovector target instruction to be fetched	1
TOTAL	11

4. Floating Point Hardware

Newer versions of UC3 CPU introduced optional floating-point hardware performing 32-bit floating-point operations. Instructions controlling this hardware are mapped into the coprocessor instruction space, addressed as coprocessor 0. The CONFIG0 system register F bit indicates if floating-point hardware is present on a specific AVR32 device.

The floating point hardware reads operands and places results in the same register file as the traditional AVR32 instructions. Floating-point compare updates the flags in the AVR32 Status Register, so that the regular AVR32 branch instructions can be used directly after a floating-point compare.

The floating-point hardware consists of a fused multiply-accumulate unit, performing $\pm A \pm (X \times Y))$ as a single operation with no intermediate rounding, thereby resulting in greater precision than if separate multiplication and addition had been performed. Hardware is also provided to convert between integer and floating-point, to compare floating-point values, and to provide initial approximations for reciprocal and reciprocal square root.

4.1 Compliance

The floating point hardware conforms to the requirements of the C standard, which is based on the IEEE 754 floating point standard.

The round-to-nearest, ties to even rounding mode is used for all instructions except float-to-integer conversions. Float-to-integer conversions use the round-to-zero mode.

The hardware supports denormal numbers.

Signalling NaN are not provided, all NaN are non-signalling (quiet). NaNs are not propagated, the default quiet NaN is always returned (0x7FC00000).

No floating-point exceptions are generated.

4.2 Operations

The floating-point instructions are mapped into the coprocessor instruction space, but use the ordinary integer register file. The ordinary integer instructions such as memory accesses and logical operations can therefore be used on the same register data as the floating point hardware uses. Therefore, no special floating-point data transfer instructions are required. All floating point instructions are mapped to coprocessor 0 *cop* instructions, i.e. they are aliases for *cop* instructions.

Attempting to execute instructions on any other coprocessor than coprocessor 0 will return a coprocessor absent exception.

Attempting to execute coprocessor 0 instructions other than *cop* on a device with floating point hardware will result in an unimplemented instruction exception.

Attempting to execute coprocessor 0 *cop* instructions on a device without floating point hardware will result in an unimplemented instruction exception.

4.2.1 Floating point compare (fcp.s)

The floating point compare instruction, *fcp.s*, updates the status register flags. Ordinary AVR32 branch instructions such as *breq* and conditional instructions such as *retge* and *movls* can use the condition flags set by *fcmp.s* directly. The following mapping from floating point compare results to AVR32 status register flags is used:

Table 4-1. Floating point compare flag setting

Compare result	Status register flags

Less	SREG[C] = 1
	SREG[N] = 1
	SREG[V] = 0
	SREG[Z] = 0

Greater	SREG[C] = 0
	SREG[N] = 0
	SREG[V] = 0
	SREG[Z] = 0

Equal	SREG[C] = 0
	SREG[N] = 0
	SREG[V] = 0
	SREG[Z] = 1

Unordered	SREG[C] = 0
	SREG[N] = 0
	SREG[V] = 1
	SREG[Z] = 0

Table 4-2. Floating point branch conditions

Branch if:	AVR32 Branch condition mnemonic
Equal	eq
Not Equal	ne
Greater than or equal	ge
Greater than	gt
Less than	lo
Less than or equal	ls
Unordered	vs

4.2.2 Floating point check (fchk.s)

This instruction checks the operand for special values, such as Not-a-Number (NaN), infinity (inf) and denormal. Status register flags are set according to the result of the *fchk.s* instruction. This instruction is useful since some algorithms require special treatment of these special values. The floating point approximation instructions updates the status register flags in the same way as *fchk.s*, since iterative approximation algorithms require special handling of these special values. Ordinary AVR32 branch instructions such as *breq* and conditional instructions such as *retge* and *movls* can use the condition flags set by *fchk.s* directly.

4.3 Instruction set

The following instructions are provided:

Table 4-3. Floating point arithmetical instructions

Mnemonics	Operands	Description	Issue latency
fmac.s	Rd, Ra, Rx, Ry	Multiply accumulate. (Rd ← Ra + Rx*Ry)	2
fnmac.s	Rd, Ra, Rx, Ry	Multiply accumulate. (Rd ← –Ra + Rx*Ry)	2
fmsc.s	Rd, Ra, Rx, Ry	Multiply subtract. (Rd ← Ra – Rx*Ry)	2
fnmsc.s	Rd, Ra, Rx, Ry	Multiply subtract. (Rd ← –Ra – Rx*Ry)	2
fmul.s	Rd, Rx, Ry	Multiply. (Rd ← Rx*Ry)	2
fnmul.s	Rd, Rx, Ry	Multiply. (Rd ← –Rx*Ry)	2
fadd.s	Rd, Rx, Ry	Add. (Rd ← Rx + Ry)	2
fsub.s	Rd, Rx, Ry	Subtract. (Rd ← Rx – Ry)	2

:

Table 4-4. Floating point conversion instructions

Mnemonics	Operands	Description	Issue latency
fcastrs.sw	Rd, Ry	Convert float to signed word, round-to-zero. (Rd ← (signed int)Ry)	1
fcastrs.uw	Rd, Ry	Convert float to unsigned word, round-to-zero. (Rd ← (unsigned int)Ry)	1
fcastsw.s	Rd, Ry	Convert signed word to float, round-to-nearest. (Rd ← (float)Ry)	1
fcastuw.s	Rd, Ry	Convert unsigned word to float, round-to-nearest. (Rd ← (float)Ry)	1

:

Table 4-5. Floating point compare instructions

Mnemonics	Operands	Description	Issue latency
fcp.s	Rd, Rx	Compare floating point values in Rd and Rx, and set status register flags accordingly.	1
fchk.s	Ry	Check floating point value in Rd for special values such as Inf, NaN and Denormal, and set status register flags accordingly.	1

:

Table 4-6. Floating point approximation instructions

Mnemonics	Operands	Description	Issue latency
frcpa.s	Rd, Ry	(Rd ← approx(1/Rx)), set status flags as *fchk.s*	1
frsqrta.s	Rd, Ry	(Rd ← approx(1/sqrt(Rx))), set status flags as *fchk.s*	1

4.4 Detailed instruction description

FMAC.S – Floating Point Multiply-Accumulate

Description

Performs multiply-accumulate of the registers specified and stores the result in destination register.

Operation:

I. Rd ← Ra + Rx*Ry;

Syntax:

I. fmac.s Rd, Ra, Rx, Ry

Operands:

I. {a, d, x, y} ∈ {0, 1, …, 15}

Status Flags:

Q:	Not affected
V:	Not affected
N:	Not affected
Z:	Not affected
C:	Not affected

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	0	0	1	1	0	1	0	Ra			

15															0
0	0	0	0	Rd				Rx				Ry			

FNMAC.S – Floating Point Negate-Multiply-Accumulate

Description

Performs negate-multiply-accumulate of the registers specified and stores the result in destination register.

Operation:

I. Rd ← - Ra + Rx*Ry;

Syntax:

I. fnmac.s Rd, Ra, Rx, Ry

Operands:

I. {a, d, x, y} ∈ {0, 1, …, 15}

Status Flags:

Q:	Not affected
V:	Not affected
N:	Not affected
Z:	Not affected
C:	Not affected

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	0	0	1	1	0	1	0	Ra			

15															0
0	0	0	1	Rd				Rx				Ry			

FMSC.S – Floating Point Multiply-Subtract

Description

Performs multiply-subtract of the registers specified and stores the result in destination register.

Operation:

I. Rd ← Ra - Rx*Ry;

Syntax:

I. fmsc.s Rd, Ra, Rx, Ry

Operands:

I. {a, d, x, y} ∈ {0, 1, …, 15}

Status Flags:

Q:	Not affected
V:	Not affected
N:	Not affected
Z:	Not affected
C:	Not affected

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	0	1	1	1	0	1	0	Ra			

15															0
0	0	0	0	Rd				Rx				Ry			

FNMSC.S – Floating Point Negate-Multiply-Subtract

Description

Performs negate-multiply-subtract of the registers specified and stores the result in destination register.

Operation:

I. Rd ← - Ra - Rx*Ry;

Syntax:

I. fnmsc.s Rd, Ra, Rx, Ry

Operands:

I. {a, d, x, y} ∈ {0, 1, …, 15}

Status Flags:

Q:	Not affected
V:	Not affected
N:	Not affected
Z:	Not affected
C:	Not affected

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	0	1	1	1	0	1	0	Ra			

15															0
0	0	0	1	Rd				Rx				Ry			

FADD.S – Floating Point Add

Description

Performs addition of the registers specified and stores the result in destination register.

Operation:

I. Rd ← Rx + Ry;

Syntax:

I. fadd.s Rd, Rx, Ry

Operands:

I. {d, x, y} ∈ {0, 1, …, 15}

Status Flags:

Q:	Not affected
V:	Not affected
N:	Not affected
Z:	Not affected
C:	Not affected

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	1	0	1	1	0	1	0	0	0	0	0

15															0
0	0	0	0	Rd				Rx				Ry			

FSUB.S – Floating Point Subtract

Description

Performs subtraction of the registers specified and stores the result in destination register.

Operation:

I. Rd ← Rx - Ry;

Syntax:

I. fsub.s Rd, Rx, Ry

Operands:

I. {d, x, y} ∈ {0, 1, …, 15}

Status Flags:

Q:	Not affected
V:	Not affected
N:	Not affected
Z:	Not affected
C:	Not affected

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	1	0	1	1	0	1	0	0	0	0	1

15						0
0	0	0	0	Rd	Rx	Ry

FMUL.S – Floating Point Multiplication

Description

Performs multiplication of the registers specified and stores the result in destination register.

Operation:

I. Rd ← Rx * Ry;

Syntax:

I. fmul.s Rd, Rx, Ry

Operands:

I. {d, x, y} ∈ {0, 1, …, 15}

Status Flags:

Q:	Not affected
V:	Not affected
N:	Not affected
Z:	Not affected
C:	Not affected

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	1	0	1	1	0	1	0	0	0	1	0

15						0
0	0	0	0	Rd	Rx	Ry

FNMUL.S – Floating Point Multiply-Negate

Description

Performs multiply-negate of the registers specified and stores the result in destination register.

Operation:

I. Rd ← - Rx * Ry;

Syntax:

I. fnmul.s Rd, Rx, Ry

Operands:

I. {d, x, y} ∈ {0, 1, …, 15}

Status Flags:

Q:	Not affected
V:	Not affected
N:	Not affected
Z:	Not affected
C:	Not affected

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	1	0	1	1	0	1	0	0	0	1	1

15						0
0	0	0	0	Rd	Rx	Ry

FCAST{S,U}W.S – Convert from Integer to Floating Point

Description

Converts the signed or unsigned integer specified and stores the result in destination register. The conversion used is rounds to nearest, ties to even.

Operation:

I. Rd ← (float)Rx;
Rx is signed integer, round-to-nearest-even

II. Rd ← (float)Rx;
Rx is unsigned integer, round-to-nearest-even

Syntax:

I.	fcastsw.s	Rd, Ry
II.	fcastuw.s	Rd, Ry

Operands:

I-IV. $\{d, y\} \in \{0, 1, \ldots, 15\}$

Status Flags:

Q:	Not affected
V:	Not affected
N:	Not affected
Z:	Not affected
C:	Not affected

Opcode:

S=0: Ry is an unsigned number, S=1: Ry is signed number

31		29	28			25	24				20	19			16
1	1	1	0	0	1	0	1	1	0	1	0	0	1	S	0

15															0
0	0	0	0	Rd				0	0	0	0	Ry			

FCASTRS.{S,U}W – Convert from Floating Point to Integer

Description

Converts the floating-point number in the specified register to a signed or unsigned integer and stores the result in destination register. Rounding used is towards zero.

Operation:

I. Rd ← (signed int)Rx;
Round towards zero

II. Rd ← (unsigned int)Rx;
Round towards zero

Syntax:

I.	fcastrs.sw	Rd, Ry
II.	fcastrs.uw	Rd, Ry

Operands:

I-IV. {d, y} ∈ {0, 1, …, 15}

Status Flags:

Q:	Not affected
V:	Not affected
N:	Not affected
Z:	Not affected
C:	Not affected

Opcode:

S=0: Ry is an unsigned number, S=1: Ry is signed number

31		29	28			25	24				20	19			16
1	1	1	0	0	1	0	1	1	0	1	0	1	0	S	1

15															0
0	0	0	0	Rd				0	0	0	0	Ry			

FCP.S – Floating Point Compare

Description

Performs a compare between the two floating point operands specified. The operation is implemented by doing a floating-point subtraction without writeback of the difference. The operation sets the status flags according to the result of the subtraction, but does not affect the operand registers. See Table 4-2, “Floating point branch conditions,” on page 41 for branch condition mnemonics corresponding to different compare results.

Operation:

I. Rx - Ry;

Syntax:

I. fcmp.s Rx, Ry

Operands:

I. {x, y} ∈ {0, 1, …, 15}

Status Flags:

Q: Not affected

Compare result	Status register flags

Less	C ← 1
	N ← 1
	V ← 0
	Z ← 0

Greater	C ← 0
	N ← 0
	V ← 0
	Z ← 0

Equal	C ← 0
	N ← 0
	V ← 0
	Z ← 1

Unordered	C ← 0
	N ← 0
	V ← 1
	Z ← 0

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	1	0	1	1	0	1	0	1	1	0	0

15															0
0	0	0	0	0	0	0	0	Rx				Ry			

FCHK.S – Floating Point Check for Special Values

Description

Checks the floating point operand specified for the special values Infinity, Not-a-Number and Denormal. A check is also performed for values with the two biggest possible representable exponents, i.e. 0xFD and 0xFE. This is useful for avoiding overflow in intermediate calculations in certain iterative algorithms. The operation sets the status flags according to the result of the check, but does not affect the operand register.

Operation:

I. Set flags depending on the value in the specified register

Syntax:

I. fchk.s Ry

Operands:

I. y ∈ {0, 1, …, 15}

Status Flags:

Q: Not affected

Predicate	Status register flag values if predicate true	Condition for branch if predicate true	Condition for branch if predicate false

Operand == NaN	C ← 1
	N ← 1
	V ← 0
	Z ← 0
		lo	gt

Operand == Infinity	C ← 0
	N ← 0
	V ← 0
	Z ← 0
		gt	lo

Operand == (Denormal or (Exponent==0xFD) or (Exponent==0xFE))	C ← 0
	N ← 0
	V ← 0
	Z ← 1
		eq	ne

Operand == Normal	C ← 0
	N ← 0
	V ← 1
	Z ← 0
		vs	vc

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	1	0	1	1	0	1	0	1	1	0	1

15															0
0	0	0	0	Rd				0	0	0	0	Ry			

FRCPA.S – Floating Point Reciprocal Approximation

Description

Returns an approximation of the reciprocal of the operand. This can be used as a starting point for iterative approximation algorithms. Also checks the operand for the special values Infinity, Not-a-Number and Denormal. A check is also performed for values with the two biggest possible representable exponents, i.e. 0xFD and 0xFE. This is useful for avoiding overflow in intermediate calculations in certain iterative algorithms. The operation sets the status flags according to the result of this check.

Operation:

I. Rd ← ApproximateReciprocal(Ry);
Set flags depending on the value in Ry

Syntax:

I. frcpa.s Rd, Ry

Operands:

I. $\{d, y\} \in \{0, 1, \ldots, 15\}$

Status Flags:

Q: Not affected

Predicate	Status register flag values if predicate true	Condition for branch if predicate true	Condition for branch if predicate false

Operand == NaN	C ← 1
	N ← 1
	V ← 0
	Z ← 0
		lo	gt

Operand == Infinity	C ← 0
	N ← 0
	V ← 0
	Z ← 0
		gt	lo

Operand == (Denormal or (Exponent==0xFD) or (Exponent==0xFE))	C ← 0
	N ← 0
	V ← 0
	Z ← 1
		eq	ne

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	1	0	1	1	0	1	0	1	1	1	0

15									0
0	0	0	0	Rd	0	0	0	0	Ry

FRSQRTA.S – Floating Point Reciprocal Square Root Approximation

Description

Returns an approximation of the reciprocal of the square root of the operand. This can be used as a starting point for iterative approximation algorithms. Also checks the operand for the special values Infinity, Not-a-Number and Denormal. A check is also performed for values with the two biggest possible representable exponents, i.e. 0xFD and 0xFE. This is useful for avoiding overflow in intermediate calculations in certain iterative algorithms. The operation sets the status flags according to the result of this check.

Operation:

I. Rd ← ApproximateSquareRootReciprocal(Ry);
Set flags depending on the value in Ry

Syntax:

I. frsqrta.s Rd, Ry

Operands:

I. {d, y} ∈ {0, 1, …, 15}

Status Flags:

Q: Not affected

Predicate	Status register flag values if predicate true	Condition for branch if predicate true	Condition for branch if predicate false

Operand == NaN	C ← 1
	N ← 1
	V ← 0
	Z ← 0
		lo	gt

Operand == Infinity	C ← 0
	N ← 0
	V ← 0
	Z ← 0
		gt	lo

Operand == (Denormal or (Exponent==0xFD) or (Exponent==0xFE))	C ← 0
	N ← 0
	V ← 0
	Z ← 1
		eq	ne

Opcode:

31		29	28			25	24				20	19			16
1	1	1	0	0	1	0	1	1	0	1	0	1	1	1	1

15															0
0	0	0	0	Rd				0	0	0	0	Ry			

5. Secure State

Revision 3 of the AVR32 architecture introduced a separate system state allowing execution of secure or secret code alongside nonsecure code on the same processor. The secret code will execute in the secure state, and therefore be protected from hacking or readout by the code executing in the nonsecure state.

Customers not needing the secure state functionality can just leave the associated hardware disabled, as it is by default, and the device will behave as previous versions of the AVR32UC.

5.1 Basic concept

The secure state architecture extension divides the memory space into two sections, a secure section and a nonsecure section. The processor can be in one of two execution states, secure or nonsecure. The SS bit in the Status Register indicates which mode the processor is in. If the processor is in the secure state, it can access both secure and nonsecure memory spaces, but if it is in the nonsecure state, only nonsecure memory sections can be accessed. The SS_ADRR and SS_ADRF registers are used to configure the sizes of these secure sections. How the SS_ADR registers map secure sections of the associated memories is determined by the individual memories, but usually SS_ADR is programmed with a secure memory size starting from the first address in the associated memory, ie. if SS_ADRF is programmed with the value 0x800, the secure section of the flash contains the addresses from 0x8000_0000 to 0x8000_07FF. Any sections of the RAM and Flash that are not in a secure section are considered nonsecure.

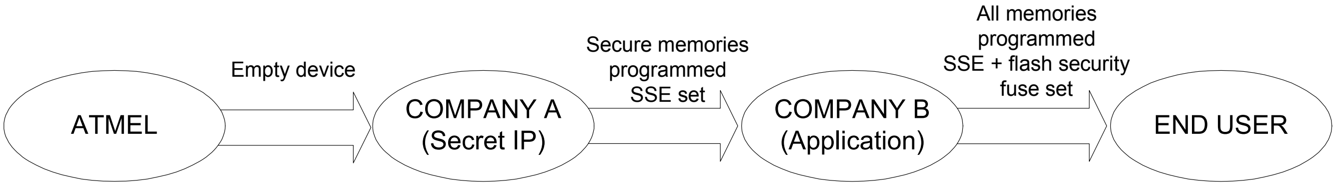
The processor can pass between the secure and nonsecure state by using dedicated *sscall* and *retss* instructions. If an access to secure memory is attempted from nonsecure space, a bus error exception is asserted and the access is aborted.

5.2 Typical use scenario

The secure state hardware support allows our customers to program their proprietary IP code such as telecom stacks, DSP libraries etc into the secure section of the memories. This secret code must be placed in a special secure section of the flash program memory, and locate its secret data structures in a special secure section of the RAM. Thereafter a dedicated fuse in non-volatile memory, called the Secure State Enable (SSE) fuse, is programmed. When set, this fuse blocks all external access to the secure memories, both from debuggers and programs running in the nonsecure sections of the processor. The SSE fuse can only be erased by a full chip erase, which will also erase all data in the memory secure sections.

This partially programmed device can then be sold to customers who will program their software application into the nonsecure section of the memories. This software can communicate with the secret IP code through a secure API provided by the secret code. This allows the application to call routines in the secret software IP, however this IP is protected from hacking or unauthorized copying. After the application has been programmed into the partially programmed device, the security fuse in the flash is set, protecting the entire application from unauthorized readout by any end user.

Figure 5-1. Typical secure state use scenario


ATMEL
Empty device
COMPANY A (Secret IP)
Secure memories programmed SSE set
COMPANY B (Application)
All memories programmed SSE + flash security fuse set
END USER

5.3 Secure state boot sequence

At system boot time, hardware state machines preloads the secure state address registers with an initial value programmed into a secure section in the flash. Also, the SS bit in the status register is preloaded with the value of the Secure State Enable (SSE) fuse from the flash. This preloading is done before the system has completed the boot sequence, so the secure state address registers and SR[SS] are initialized before code starts executing and before the debug system has been enabled.

5.4 Secure state debugging

Normally, debugging when executing in secure state should be turned off to prevent compromising the secure code. However, it is useful to allow debugging of the secure state code during development of this code. A fuse in flash, called Secure State Debug Enable (SSDE), can be programmed to enable debugging of secure state code.

5.5 Events in secure state

Normal RISC state interrupt and exception handling has been described in Section 3.7 ”Event handling” on page 22. This behavior is modified in the following way when interrupts and exceptions are received in secure state:

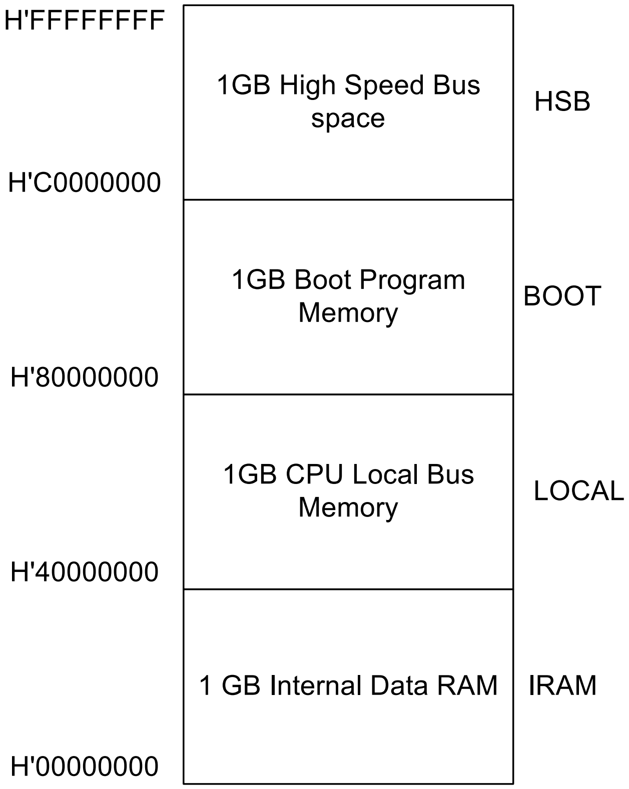
- A *sscall* instruction will set SR[GM]. In secure state, SR[GM] masks both INT0-INT3, and NMI. Clearing SR[GM], INT0-INT3 and NMI will remove the mask of these event sources. INT0-INT3 are still additionally masked by the I0M-I3M bits in the status register.
- *sscall* has handler address at 0x8000_0004.
- Exceptions have a handler address at 0x8000_0008.
- NMI has a handler address at 0x8000_000C.
- BREAKPOINT has a handler address at 0x8000_0010.
- INT0-INT3 are not autovectored, but have a common handler address at 0x8000_0014.

Note that in the secure state, all exception sources share the same handler address. It is therefore not possible to separate different exception causes when in the secure world. The secure world system must be designed to support this, the most obvious solution is to design the secure software so that exceptions will not arise when executing in the secure world.

6. Memory System

AVR32UC implements a 32-bit unsegmented memory space. Regions of this memory space can be protected by an optional MPU. The memory map is as follows:

Figure 6-1. The AVR32UC memory map.



6.1 Memory sections

The memory map contains four sections, named IRAM, LOCAL, BOOT and HSB. The IRAM section contains the internal EX stage memory, and this memory is mapped from address 0 and upwards. The LOCAL section is mapped from address 0x4000_0000 and is designed for containing device-specific high-speed interfaces, such as floating-point units, encryption hardware or high-speed GPIO ports. Access to the LOCAL space is performed using any ordinary load and store instructions, and is performed in a single clock cycle. Mapping timing-critical devices in the LOCAL section is beneficial as the interface operates with high clock frequency, and its timing is deterministic since it does not need to access a shared bus which may be heavily loaded.

The BOOT section starts at address 0x8000_0000, which is the reset address for AVR32UC. This section will typically contain an internal program FLASH, mapped from address 0x8000_0000 and upwards. The HSB section contains the addresses of all modules mapped on the HSB bus. This may include peripherals such as USARTs and external memory interfaces.

The memory space is uniform, so program code can execute from the IRAM, BOOT and HSB sections, and data accesses can be performed to any of the these sections. Note that implementations of AVR32UC of may forbid certain accesses to certain memory sections, eg a write to program FLASH mapped into the BOOT section may be forbidden. The LOCAL section is only accessible by the Load-Store Unit in the CPU EX pipeline stage, therefore, code can not be executed from addresses in the LOCAL space.

6.2 Memory interfaces

The AVR32UC CPU has three memory interfaces:

- IF stage HSB master interface for instruction fetches
- EX stage HSB master interface for data accesses into BOOT or HSB sections
- EX stage HSB slave interface enabling other parts of the system to access addresses in the IRAM section

6.3 IF stage interface

The single master interface in the IF stage performs instruction fetches. All fetches are performed with word alignment, except for the first fetch after a change-of-flow, which may use halfword alignment. The IF stage can not perform writes, only reads are possible. Reads can be perfomed from all addresses mapped on the HSB bus. Reads are performed as incrementing bursts of unspecified length. The IF stage master interface will stall appropriately to support slow slaves.

6.4 EX stage interfaces

The EX stage separates between CPU accesses to the IRAM section, and accesses to BOOT/HSB. Any access to the IRAM section are performed to dedicated, high-speed RAMs implemented inside the memory controller. These fast RAMs are able to read or write within the cycle they are initiated. This means that a load instruction in EX will have the read-data ready at the end of the clock cycle for writing into the register file.

6.4.1 EX stage HSB master interface

Any CPU access to the BOOT or HSB sections will use multiple clock cycles, as dictated by the HSB semantics. Writes to the BOOT or HSB sections can be pipelined, and are performed as a stream of nonsequential transfers, each taking one cycle unless stalled by the slave. If the slave stalls the transfer, the CPU will stall until the slave releases the stall. CPU reads from the BOOT or HSB sections are not pipelined, and transfer of a data therefore takes two clock cycles, one cycle for the address phase, and one cycle for the data phase. The CPU will be stalled in the data phase.

6.4.2 EX stage HSB slave interface

The AVR32UC CPU provides a slave interface into the high-speed RAMs that are implemented inside the memory controller. This interface enables other parts of the system, like DMA controllers, to write or read data to or from the RAMs. The slave interface support bursts for both reads and writes. If the high-speed RAMs for some reason cannot accept the transfer request, it will reply by stalling the request until it can be serviced.

The arbitration priorities between the CPU and the slave interface for the RAMs can be controlled by programming the CPU Control Register (CPUCR). The CPUCR is described in Section 2.5 on page 11. Arbitration is performed according to the following rules:

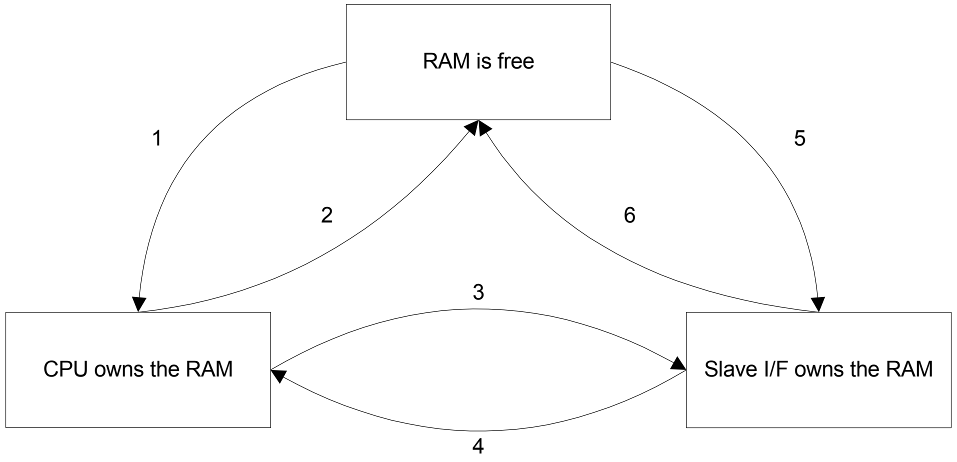
Assuming the memory interface is idle, and no memory transfers have been performed. Whoever requests access to the RAMs will win the arbitration and get access. If both the CPU and the slave interface requests access, the CPU will win.

The source that won the arbitration can use the RAMs for as long as they require. If the other source also has a pending request for use of the RAM, this source will have to wait maximum the number of cycles specified by the SPL or CPL fields of CPUCR. The pending source will gain access to the RAMs when the current owner voluntarily releases the RAMs, or after the SPL/CPL timeout period, whichever comes first.

If the CPU wins arbitration for the RAMs, the CPU is guaranteed to own the RAM for the period specified by the COP field in CPUCR. Any slave request will be left pending during this period, even if the CPU is not using the RAMs.

The following state diagram shows the states in arbitration for the RAM.

Figure 6-2. Arbitration between CPU and slave interface for RAMs.



The state transitions are as follows:

```
1: CPU_wants_to_perform_mem_access
2: CPU_access_complete && (been_in_state > CPUCR[COP])
3: (been_in_state > CPUCR[COP]) && slave_wants_to_perform_mem_access &&
(slave_been_pending > CPUCR[SPL])
4: CPU_wants_to_perform_mem_access && (CPU_been_pending > CPUCR[CPL])
5: slave_wants_to_perform_mem_access && !CPU_wants_to_perform_mem_access
6: slave_access_complete
```

6.4.3 EX stage local bus interface

Any CPU access to the the LOCAL section is completed in a single clock cycle, both for reads and writes. Transfers on this bus can not be stalled. The CPU will never be stalled due to an access to the LOCAL section. Accesses to this section is performed using regular load-store instructions such as for example *ldswp.w*, *ld.w*, *ld.ub*, *st.w*, *stswp.w*, *ldm* or *stm*.

Which devices are mapped in the LOCAL section, and their memory maps, is device-specific.

The LOCAL interface must be enabled by the user by programming the LOCEN bit in CPUCR. Accesses to LOCAL memory addresses without first enabling the section will result in a BUS ERROR exception.

If the MPU is enabled, accesses to LOCAL will be subject to permission checking.

To ensure maximum transfer speed and cycle determinism, any slaves being addressed by the CPU on the local bus must be able to receive and transmit data on the bus at CPU clock speeds. The consequences of this may vary between different slave devices, but for some slave devices it may imply that the slaves have to run at the CPU clock frequency when local bus transfers are being performed. Refer to the device datasheet for information on any relationships between CPU and device clock frequencies imposed by the local bus.

6.5 IRAM Write buffer

The EX stage has a write buffer used to hold data to be written to the IRAM section. The operation of this buffer is usually transparent to the programmer. The programmer should be aware of the following:

- The IRAM has a single port, allowing either one read or one write per clock cycle.
- The write buffer is pipelined, allowing sequential writes to IRAM to be pipelined without any pipeline stalls. The previous contents of the write buffer is written to the RAM in parallel with the new store data being placed in the write buffer.
- Any read instruction to IRAM in EX will be performed immediately, even if a previous store instruction has placed data to store in the write buffer. In this case, the previous store data remains in the write buffer and will be written back to RAM in a later clock cycle.
- If a read instruction in EX accesses the same address as the data in the write buffer is to be stored to, the pipeline is stalled for one clock cycle while the write buffer is emptied to RAM. The read will be performed normally in the next clock cycle.
- The contents of the write buffer is written to the physical RAM as soon as the memory interface is not used by any instructions.
- The state of the write buffer may affect the timing of RMW instructions, see "Read-modify-write instructions" on page 84 for details

6.6 Memory barriers

Memory barriers are constructs used to enfore memory consitency. Caches and self-modifying code may cause memory to become inconsistent. AVR32UC has a simple pipeline with no caches, so there is usually no need for memory barriers. Mechanisms for memory barriers are present to handle the cases where such barriers are needed.

6.6.1 Instruction memory barriers

An instruction memory barrier (IMB) is usually only needed when executing self-modifying code, for example when self-programming program flash. In this case, one must ensure that all levels in the memory hierarchy are consistent. Due to the simple non-cached memory system in AVR32UC, this is usually trivial.

The programmer should make sure that an IMB is used if there is a possibility that an instruction to be modified by self-modifying code has already been prefetched by the instruction prefetch unit. In this case, an IMB should be inserted between the instruction modifying the code and the execution of the modified instruction. To make sure that the modified version of the instruction is executed, the prefetch buffer should be flushed between changing the program memory and executing the new version of the program.

Any instruction performing a change-of flow, such as return from exception, conditional branches, unconditional branches, subprogram call or return, or instructions writing to PC would implement an IMB in AVR32UC.

6.6.2 Data memory barriers

A data memory barrier (DMB) is used to make sure that a data memory access, either a read or write, is actually performed before the rest of the code is executed. Caches, write buffers and bus latency may cause a memory access to be seen by a slave many cycles after it has been executed by the pipeline. In some cases, this may lead to UNPREDICTABLE behavior in the system.

One example of this is found in interrupt handlers. One usually wants to make sure that the interrupt request has been cleared before executing the *rete* instruction, otherwise the same interrupt may be serviced immediately after executing the *rete* instruction. In this case a DMB must be inserted between the code clearing the interrupt request and the *rete*.

All accesses to HSB space are strongly ordered. This is used to implement DMBs. A DMB after a store to a HSB slave is implemented by performing a dummy read from the same slave. Any critical code after the read will stall until the read has been performed.

Consider an interrupt request made by a peripheral. This peripheral will disassert the interrupt request as soon as the interrupt handler has written a specific bitmask to its PERIPH_INTCLEAR register. A read from the same peripheral performs a bus transfer that implements the DMB. The rete instruction can be executed after the DMB.

Code 6-1. Clearing IRQs using data memory barriers

```
// Using data memory barriers in the IRQ handler to make sure that the
// request has been disasserted before returning from the handler
// Assume that the IRQ is cleared by writing a bitmask to PERIPH_INTCLEAR.
// r0 points to this register, r1 contains the correct bitmask.
irq_handler:
    <some instructions>
    st.w r0[0], r1
    ld.w r12, r0[0] // data memory barrier
    rete
```

7. Memory Protection Unit

The AVR32 architecture defines an optional Memory Protection Unit (MPU). This is a simpler alternative to a full MMU, while at the same time allowing memory protection. The MPU allows the user to divide the memory space into different protection regions. These protection regions have a user-defined size, and starts at a user-defined address. The different regions can have different cacheability attributes and bufferability attributes. Each region is divided into 16 subregions, each of these subregions can have one of two possible sets of access permissions.

The MPU does not perform any address translation.

7.1 Memory map in systems with MPU

An AVR32 implemetation with a MPU has a flat, unsegmented memory space. Access permissions are given only by the different protection regions.

7.2 Understanding the MPU

The AVR32 Memory Protection Unit (MPU) is responsible for checking that memory transfers have the correct permissions to complete. If a memory access with unsatisfactory privileges is attempted, an exception is generated and the access is aborted. If an access to a memory address that does not reside in any protection region is attempted, an exception is generated and the access is aborted.

The user is able to allow different privilege levels to different blocks of memory by configuring a set of registers. Each such block is called a protection region. Each region has a user-programmable start address and size. The MPU allows the user to program 8 different protection regions. Each of these regions have 16 sub-regions, which can have different access permissions, cacheability and bufferability.

The “DMMU SZ” fields in the CONFIG1 system register identifies the number of implemented protection regions, and therefore also the number of MPU registers. An AVR32UC system with caches also have MPU cacheability and bufferability registers.

A protection region can be from 4 KB to 4 GB in size, and the size must be a power of two. All regions must have a start address that is aligned to an address corresponding to the size of the region. If the region has a size of 8 KB, the 13 lowest bits in the start address must be 0. Failing to do so will result in UNDEFINED behaviour. Since each region is divided into 16 sub-regions, each sub-region is 256 B to 256 MB in size.

When an access hits into a memory region set up by the MPU, hardware proceeds to determine which subregion the access hits into. This information is used to determine whether the access permissions for the subregion are given in MPUAPRA/MPUBRA/MPUCRA or in MPUAPRB/MPUBRB/MPUCRB.

If an access does not hit in any region, the transfer is aborted and an exception is generated.

The MPU is enabled by writing setting the E bit in the MPUCR register. The E bit is cleared after reset. If the MPU is disabled, all accesses are treated as uncacheable, unbufferable and will not generate any access violations. Before setting the E bit, at least one valid protection region must be defined.

7.2.1 MPU interface registers

The following registers are used to control the MPU, and provide the interface between the MPU and the operating system, see Figure 7-1 on page 67. All the registers are mapped into the System Register space, their addresses are presented in "System registers" on page 11. They are accessed with the *mtsr* and *mfsr* instructions.

The MPU interface registers are shown below. The suffix n can have the range 0-7, indicating which region the register is associated with.

Figure 7-1. The MPU interface registers

MPUARn

31 12	11 6	5 1	0
Base Address	-	Size	V

MPUPSRn

31 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

MPUCRA / MPUCRB

31 8	7	6	5	4	3	2	1	0
-	C7	C6	C5	C4	C3	C2	C1	C0

MPUBRA / MPUBRB

31 8	7	6	5	4	3	2	1	0
-	B7	B6	B5	B4	B3	B2	B1	B0

MPUAPRA / MPUAPRB

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
AP7	AP6	AP5	AP4	AP3	AP2	AP1	AP0

MPUCR

31 1	0
-	E

7.2.1.1 *MPU Address Register - MPUARn*

A MPU Address register is implemented for each of the 8 protection regions. The MPUAR registers specify the start address and size of the regions. The start address must be aligned so that its alignment corresponds to the size of the region. The minimum allowable size of a region is 4 KB, so only bits 31:12 in the base address needs to be specified. The other bits are always 0. Each MPUAR also has a valid bit that specifies if the protection region is valid. Only valid regions are considered in the protection testing.

The MPUAR register consists of the following fields:

- Base address - The start address of the region. The minimum size of a region is 4KB, so only the 20 most significant bits in the base address needs to be specified. The 12 lowermost base address bits are implicitly set to 0. If protection regions larger than 4 KB is used, the user must write the appropriate bits in Base address to 0, so that the base address is aligned to the size of the region. Otherwise, the result is UNDEFINED.

- Size - Size of the protection region. The possible sizes are shown in Table 7-1 on page 68.

Table 7-1. Protection region sizes implied by the Size field

Size	Region size	Constraints on Base address
B'00000 to B'01010	UNDEFINED	-
B'01011	4 KB	None
B'01100	8 KB	Bit [12] in Base Address must be 0
B'01101	16 KB	Bit [13:12] in Base Address must be 0
B'01110	32 KB	Bit [14:12] in Base Address must be 0
B'01111	64 KB	Bit [15:12] in Base Address must be 0
B'10000	128 KB	Bit [16:12] in Base Address must be 0
B'10001	256 KB	Bit [17:12] in Base Address must be 0
B'10010	512 KB	Bit [18:12] in Base Address must be 0
B'10011	1 Mb	Bit [19:12] in Base Address must be 0
B'10100	2 MB	Bit [20:12] in Base Address must be 0
B'10101	4 MB	Bit [21:12] in Base Address must be 0
B'10110	8 MB	Bit [22:12] in Base Address must be 0
B'10111	16 MB	Bit [23:12] in Base Address must be 0
B'11000	32 MB	Bit [24:12] in Base Address must be 0
B'11001	64 MB	Bit [25:12] in Base Address must be 0
B'11010	128 MB	Bit [26:12] in Base Address must be 0
B'11011	256 MB	Bit [27:12] in Base Address must be 0
B'11100	512 MB	Bit [28:12] in Base Address must be 0
B'11101	1 GB	Bit [29:12] in Base Address must be 0
B'11110	2 GB	Bit [30:12] in Base Address must be 0
B'11111	4 GB	Bit [31:12] in Base Address must be 0

- V - Valid. Set if the protection region is valid, cleared otherwise. This bit is written to 0 by a reset. The region is not considered in the protection testing if the V bit is cleared.

7.2.1.2 MPU Permission Select Register - MPUPSRn

A MPU Permission Select register is implemented for each of the 8 protection regions. Each MPUPSR register divides the protection region into 16 subregions. The bitfields in MPUPSR specifies whether each subregion has access permissions as specified by the region entry in either MPUAPRA or MPUAPRB.

Table 7-2. Subregion access permission implied by MPUPSR bitfields

MPUPSRn[P]	Access permission
0	MPUAPRA[APn]
1	MPUAPRB[APn]

7.2.1.3 *MPU Cacheable Register A / B- MPUCRA / MPUCRB*

The MPUCR registers have one bit per region, indicating if the region is cacheable. If the corresponding bit is set, the region is cacheable. The register is written to 0 upon reset.

AVR32UC implementations may optionally choose not to implement the MPUCR registers.

7.2.1.4 *MPU Bufferable Register A / B- MPUBRA / MPUBRB*

The MPUBR registers have one bit per region, indicating if the region is bufferable. If the corresponding bit is set, the region is bufferable. The register is written to 0 upon reset.

AVR32UC implementations may optionally choose not to implement the MPUBR registers.

7.2.1.5 *MPU Access Permission Register A / B - MPUAPRA / MPUAPRB*

The MPUAPR registers indicate the access permissions for each region. The MPUAPR is written to 0 upon reset. The possible access permissions are shown in Table 7-3 on page 69.

Table 7-3. Access permissions implied by the APn bits

AP	Privileged mode	Unprivileged mode
B’0000	Read	None
B’0001	Read / Execute	None
B’0010	Read / Write	None
B’0011	Read / Write / Execute	None
B’0100	Read	Read
B’0101	Read / Execute	Read / Execute
B’0110	Read / Write	Read / Write
B’0111	Read / Write / Execute	Read / Write / Execute
B’1000	Read / Write	Read
B’1001	Read / Write	Read / Execute
B’1010	None	None
Other	UNDEFINED	UNDEFINED

7.2.1.6 *MPU Control Register - MPUCR*

The MPUCR controls the operation of the MPU. The MPUCR has only one field:

- E - Enable. If set, the MPU address checking is enabled. If cleared, the MPU address checking is disabled and no exceptions will be generated by the MPU.

7.2.2 MPU exception handling

This chapter describes the exceptions that can be signalled by the MPU.

7.2.2.1 *ITLB Protection Violation*

An ITLB protection violation is issued if an instruction fetch violates access permissions. The violating instruction is not executed. The address of the failing instruction is placed on the system stack.

7.2.2.2 DTLB Protection Violation

An DTLB protection violation is issued if a data access violates access permissions. The violating access is not executed. The address of the failing instruction is placed on the system stack.

7.2.2.3 ITLB Miss Violation

An ITLB miss violation is issued if an instruction fetch does not hit in any region. The violating instruction is not executed. The address of the failing instruction is placed on the system stack.

7.2.2.4 DTLB Miss Violation

An DTLB miss violation is issued if a data access does not hit in any region. The violating access is not executed. The address of the failing instruction is placed on the system stack.

7.2.2.5 TLB Multiple Hit Violation

An access hit in multiple protection regions. The address of the failing instruction is placed on the system stack. This is a critical system error that should not occur.

7.3 Example of MPU functionality

As an example, consider region 0. Let region 0 be of size 16 KB, thus each subregion is 1KB. Subregion 0 has offset 0-1KB from the base address, subregion 1 has offset 1KB-2KB and so on.

MPUAPRA and MPUAPRB each has one field per region. Each subregion in region 0 can get its access permissions from either MPUAPRA[AP0] or MPUAPRB[AP0], this is selected by the subregion's bitfield in MPUPSR0.

Let:

MPUPSR0 = {0b0000_0000_0000_0000, 0b1010_0000_1111_0101}

MPUAPRA = {A, B, C, D, E, F, G, H}

MPUAPRB = {a, b, c, d, e, f, g, h}

where {A-H, a-h} have legal values as defined in Table 7-3.

Thus for region 0:

Table 7-4. Example of access rights for subregions

Subregion	Access permission	Subregion	Access permission
0	h	8	H
1	H	9	H
2	h	10	H
3	H	11	H
4	h	12	H
5	h	13	h
6	h	14	H
7	h	15	h

8. Instruction Cycle Summary

This chapter presents the instructions in AVR32UC CPU, and the number of clock cycle they require to complete. All the instructions in each group behave similarly in the pipeline. The final subchapter presents code examples to illustrate the clock cycle requirements of various code constructs.

8.1 Definitions

The following definitions are presented in the tables below:

8.1.1 Issue

An instruction is *issued* when it leaves the ID stage and enters the EX stage.

8.1.2 Issue latency

The *issue latency* represents the number of clock cycles required between the issue of the instruction and the issue of the following instruction. For some change-of-flow instructions, this includes the cycle penalty caused by the pipeline flush. The issue latency assumes, unless stated otherwise, that the instruction and data memories are able to return an instruction or data in a single cycle, which may not be true for slow program memories or data memories mapped on the HSB bus.

8.2 Special considerations

8.2.1 PC as destination register

Most instructions can take PC as destination register. This will result in a jump to the calculated address. The jump is performed when the instruction writing to PC has completed, and all other effects of the instruction, like updating of pointer registers for load instructions with PC as target instruction, have been committed. Instructions writing to PC will have an additional issue latency of 2 cycles due to the pipeline flush.

8.2.2 Alignment of change-of-flow targets

The cycle count number for change-of-flow instructions assumes that the target instruction is a compact instruction or word-aligned extended instruction. An extra cycle will be required if the target instruction is a halfword-aligned extended instruction, since both halves of the instruction must be fetched before it can be issued.

8.2.3 Memory and bus timings

Performance of memory accesses and instruction fetching are affected by the performance of system memories and system bus. The following are examples of factors that may affect the cycle count of such operations:

- Accesses to the IRAM section in parallel with another bus master, for example a DMA controller.
- Accesses to memories with wait states, for example flash or external memories.
- Using system buses with wait states or arbitration overhead.
- Accesses to memories that are simultaneously being accessed by other bus masters.

8.3 CPU revision

Revision 1, 2 and 3 of the AVR32UC CPU has the same instruction timings, except that the divider in revision 2 and later is faster than in revision 1. Instructions only present in revision 2 or 3 of the CPU are explicitly noted.

8.4 ALU instructions

This group comprises simple single-cycle ALU instructions like add and sub. The conditional subtract and move instructions are also in this group. All instructions in this group, except *ssrf* to bits 15 to 31, take one cycle to execute, and the result is available for use by the following instruction.

Table 8-1. ALU instructions

Mnemonics		Operands	Description	Issue latency
abs	C	Rd	Absolute value.	1
acr	C	Rd	Add carry to register.	1
adc	E	Rd, Rx, Ry	Add with carry.	1
add	C	Rd, Rs	Add.	1
	E	Rd, Rx, (Ry << sa)	Add shifted.	1
add{cond4}	E	Rd, Rx, Ry	Add if condition satisfied. CPU revision 2 and higher only.	1
addhh.w	C	Rd, Rx<part>, Ry<part>	Add signed halfwords (32 ← 16 + 16)	1
addabs	E	Rd, Rx, Ry	Add with absolute value.	1
cp.b	E	Rd, Rs	Compare byte.	1
cp.h	E	Rd, Rs	Compare halfword.	1
cp.w	C	Rd, Rs	Compare.	1
	C	Rd, imm		1
	E	Rd, imm		1
cpc	C	Rd	Compare with carry.	1
	E	Rd, Rs		1
max	E	Rd, Rx, Ry	Return signed maximum	1
min	E	Rd, Rx, Ry	Return signed minimum	1
neg	C	Rd	Two's Complement.	1
rsub	C	Rd, Rs	Reverse subtract.	1
	E	Rd, Rs, k8		1
rsub{cond4}	E	Rd, imm	Reverse subtract immediate if condition satisfied. CPU revision 2 and higher only.	1
sbc	E	Rd, Rx, Ry	Subtract with carry.	1
scr	C	Rd	Subtract carry from register.	1

Table 8-1. ALU instructions

sub	C	Rd, Rs	Subtract.	1
	E	Rd, Rx, (Ry << sa)		1
	C	Rd, imm		1
	E	Rd, imm		1
	E	Rd, Rs, imm		1
subhh.w	C	Rd, Rx<part>, Ry<part>	Subtract signed halfwords (32 ← 16 - 16)	1
sub{cond4}	E	Rd, imm	Subtract immediate if condition satisfied.	1
	E	Rd, imm	Subtract if condition satisfied. CPU revision 2 and higher only.	1
tnbz	C	Rd	Test no byte equal to zero.	1
and	C	Rd, Rs	Logical AND.	1
	E	Rd, Rx, Ry << sa		1
	E	Rd, Rx, Ry >> sa		1
and{cond4}	E	Rd, Rx, Ry	Logical AND if condition satisfied. CPU revision 2 and higher only.	1
andn	C	Rd, Rs	Logical AND NOT.	1
andh	E	Rd, imm	Logical AND High Halfword, low halfword is unchanged.	1
	E	Rd, imm, COH	Logical AND High Halfword, clear other halfword.	1
andl	E	Rd, imm	Logical AND Low Halfword, high halfword is unchanged.	1
	E	Rd, imm, COH	Logical AND Low Halfword, clear other halfword.	1
com	C	Rd	One's Complement (NOT).	1
eor	C	Rd, Rs	Logical Exclusive OR.	1
	E	Rd, Rx, Ry << sa		1
	E	Rd, Rx, Ry >> sa		1
eor{cond4}	E	Rd, Rx, Ry	Logical EOR if condition satisfied. CPU revision 2 and higher only.	1
eorh	E	Rd, imm	Logical Exclusive OR (High Halfword).	1
eorl	E	Rd, imm	Logical Exclusive OR (Low Halfword).	1
or	C	Rd, Rs	Logical (Inclusive) OR.	1
	E	Rd, Rx, Ry << sa		1
	E	Rd, Rx, Ry >> sa		1
or{cond4}	E	Rd, Rx, Ry	Logical OR if condition satisfied. CPU revision 2 and higher only.	1

Table 8-1. ALU instructions

orh	E	Rd, imm	Logical OR (High Halfword).	1
orl	E	Rd, imm	Logical OR (Low Halfword).	1
tst	C	Rd, Rs	Test register for zero.	1
bfins	E	Rd, Rs, o5, w5	Insert the lower w5 bits of Rs in Rd at bit-offset o5.	1
bfexts	E	Rd, Rs, o5, w5	Extract and sign-extend the w5 bits in Rs starting at bit-offset o5 to Rd.	1
bfextu	E	Rd, Rs, o5, w5	Extract and zero-extend the w5 bits in Rs starting at bit-offset o5 to Rd.	1
bld	E	Rd, b5	Bit load.	1
brev	C	Rd	Bit reverse.	1
bst	E	Rd, b5	Bit store.	1
casts.b	C	Rd	Typecast byte to signed word.	1
casts.h	C	Rd	Typecast halfword to signed word.	1
castu.b	C	Rd	Typecast byte to unsigned word.	1
castu.h	C	Rd	Typecast halfword to unsigned word.	1
cbr	C	Rd, b5	Clear bit in register.	1
clz	E	Rd, Rs	Count leading zeros.	1
sbr	C	Rd, b5	Set bit in register.	1
swap.b	C	Rd	Swap bytes in register.	1
swap.bh	C	Rd	Swap bytes in each halfword.	1
swap.h	C	Rd	Swap halfwords in register.	1
asr	E	Rd, Rx, Ry	Arithmetic shift right (signed).	1
	E	Rd, Rs, sa		1
	C	Rd, sa		1
lsl	E	Rd, Rx, Ry	Logical shift left.	1
	E	Rd, Rs, sa		1
	C	Rd, sa		1
lsr	E	Rd, Rx, Ry	Logical shift right.	1
	E	Rd, Rs, sa		1
	C	Rd, sa		1
rol	C	Rd	Rotate left through carry.	1
ror	C	Rd	Rotate right through carry.	1
mov	C	Rd, imm	Load immediate into register.	1
	E	Rd, imm		1
	C	Rd, Rs	Copy register.	1
mov{cond4}	E	Rd, Rs	Copy register if condition is true	1
	E	Rd, imm	Load immediate into register if condition is true	1

Table 8-1. ALU instructions

movh	E	Rd, imm	Load immediate into high halfword of register. CPU revision 2 and higher only.	1
csrf	C	b5	Clear status register flag.	1
csrfcz	C	b5	Copy status register flag to C and Z.	1
ssrf	C	b5	Set status register flag.	1 / 3
sr{cond4}	C	Rd	Conditionally set register to true or false	1

8.5 Multiply instructions

These instructions require one pass through the multiplier array and produce a 32- or 48-bit result. For *mulrndhh*, a rounding value of 0x8000 is added to the product producing the final result. This group does not set any flags, except for the *mulsat* instructions which set Q if saturation occurred.

Table 8-2. Multiply instructions

Mnemonics		Operands	Description	Issue latency
mul	E	Rd, Rx, Ry	Multiply. (32 ← 32 x 32)	1
	E	Rd, Rs, imm	Multiply immediate.	1
mulhh.w	E	Rd, Rx<part>, Ry<part>	Signed Multiply of halfwords. (32 ← 16 x 16)	1
mulnhh.w	E	Rd, Rx<part>, Ry<part>	Signed Multiply of halfwords. (32 ← 16 x 16)	1
mulnwh.d	E	Rd, Rx, Ry<part>	Signed Multiply, word and halfword. (48 ← 32 x 16)	1
mulwh.d	E	Rd, Rx, Ry<part>	Signed Multiply, word and halfword. (48 ← 32 x 16)	1
mulsathh.h	E	Rd, Rx<part>, Ry<part>	Fractional signed multiply with saturation. Return halfword. (16 ← 16 x 16)	1
mulsathh.w	E	Rd, Rx<part>, Ry<part>	Fractional signed multiply with saturation. Return word. (32 ← 16 x 16)	1
mulsatwh.w	E	Rd, Rx, Ry<part>	Fractional signed multiply with saturation. Return word. (32 ← 32 x 16)	1
mulsatrndhh.h	E	Rd, Rx<part>, Ry<part>	Signed multiply with rounding. Return halfword. (16 ← 16 x 16)	1
mulsatrndwh.w	E	Rd, Rx, Ry<part>	Signed multiply with rounding. Return halfword. (32 ← 32 x 16)	1

8.6 MAC instructions

These instructions require one pass through the multiplier array and produce a 32- or 48-bit result. This result is added to an accumulator register. A valid copy of this accumulator may be cached in the accumulator cache. Otherwise, an extra cycle is needed to read the accumulator from the register file. Therefore, issue and result latencies depend on whether the accumulator is cached in the AccCache.

This group does not set any flags, except for the *macsathh.w* instruction which set Q if saturation occurred.

Table 8-3. MAC instructions

Mnemonics		Operands	Description	Issue latency
mac	E	Rd, Rx, Ry	Multiply accumulate. (32 ← 32x32 + 32)	1/2
machh.w	E	Rd, Rx<part>, Ry<part>	Multiply signed halfwords and accumulate. (32 ← 16x16 + 32)	1/2
machh.d	E	Rd, Rx<part>, Ry<part>	Multiply signed halfwords and accumulate. (48 ← 16x16 + 48)	1/2
macwh.d	E	Rd, Rx, Ry<part>	Multiply signed word and halfword and accumulate. (48 ← 32 x 16 + 48)	1/2
macsathh.w	E	Rd, Rx<part>, Ry<part>	Fractional signed multiply accumulate with saturation. Return word. (32 ← 16 x 16 + 32)	1/2

8.7 MulMac64 instructions

These instructions require two passes through the multiplier array to produce a 64-bit result. For *macs.d* and *macu.d*, a valid copy of this accumulator may be cached in the accumulator cache. Otherwise, an extra cycle is needed to read the accumulator from the register file. Therefore, issue and result latencies depend on whether a valid entry is found in the accumulator cache.

Table 8-4. MulMac64 instructions

Mnemonics		Operands	Description	Issue latency
macs.d	E	Rd, Rx, Ry	Multiply signed accumulate. (64 ← 32x32 + 64)	3/4
macu.d	E	Rd, Rx, Ry	Multiply unsigned accumulate. (64 ← 32x32 + 64)	3/4
muls.d	E	Rd, Rx, Ry	Signed Multiply. (64 ← 32 x 32)	2
mulu.d	E	Rd, Rx, Ry	Unsigned Multiply. (64 ← 32 x 32)	2

8.8 Divide instructions

These instructions require several cycles in the EX stage to complete. The *divs* and *divu* instructions will be aborted immediately if any interrupts are pending, in order to limit the interrupt latency. The divide instructions are faster in revision 2 than in revision 1 of the AVR32UC CPU.

Table 8-5. Divide instructions

Mnemonics		Operands	Description	Issue latency
divs	E	Rd, Rx, Ry	Divide signed. (32 ← 32/32) (32 ← 32%32)	19[1]
divu	E	Rd, Rx, Ry	Divide unsigned. (32 ← 32/32) (32 ← 32%32)	19[1]

1.) 35 cycles in revision 1 of the CPU

8.9 Saturate instructions

These instructions perform arithmetic operations with possible saturation.

Table 8-6. Saturate instructions

Mnemonics		Operands	Description	Issue latency
satadd.h	E	Rd, Rx, Ry	Saturated add halfwords.	1
satadd.w	E	Rd, Rx, Ry	Saturated add.	1
satsub.h	E	Rd, Rx, Ry	Saturated subtract halfwords.	1
satsub.w	E	Rd, Rx, Ry	Saturated subtract.	1
	E	Rd, Rs, imm		1
satrnds	E	Rd >> sa, b5	Signed saturate from bit given by sa after a right shift with rounding of b5 bit positions.	2
satrndu	E	Rd >> sa, b5	Unsigned saturate from bit given by sa after a right shift with rounding of b5 bit positions.	2
sats	E	Rd >> sa, b5	Shift sa positions and do signed saturate from bit given by b5.	1
satu	E	Rd >> sa, b5	Shift sa positions and do unsigned saturate from bit given by b5.	1

8.10 Load and store instructions

This group includes all the load and store instructions. The address calculations are performed by the adder in the EX stage. The EX adder also performs the writeback address calculation for the autoincrement and autodecrement operation.

Loaded data are available at the end of the cycle in the EX stage. Byte and halfword data must be extended and rotated before they are valid. This is performed in the EX stage. *Ldins* and *ldswp* instructions also require modification in the EX stage before their results are valid. *Stswp* instructions require modification before their data is output to the memory interface. This modification is performed in the EX stage.

The *stcond* instruction takes 2 cycles if the store is not performed, 3 cycles if the store is performed.

All issue latencies are given for accesses to IRAM or LOCAL. These timings must be modified as follows for accesses to BOOT or HSB sections:

- A byte, halfword or word load requires 1+w cycles in addition to the count listed in Table 8-7, where w is the number of wait states from the slave and bus system. The pipeline will stall during these cycles.
- A doubleword load performs two memory accesses, so 2(1+w) cycles are needed in addition to the count listed in Table 8-7. The pipeline will stall during these cycles.
- A byte, halfword or word store requires (1+w) cycles in addition to the count listed in Table 8-7, where w is the number of wait states from the slave and bus system. Stores to BOOT or HSB can be performed in the background, so the pipeline will only stall if another memory access is attempted during these w cycles. However, multiple stores to addresses in BOOT or HSB can be automatically combined by the memory interface to create bursts on the HSB bus. This means that any consecutive stores to BOOT or HSB sections will not stall the pipeline unless the bus itself inserts wait cycles, for example due to wait states or bus contention. Instructions not performing memory accesses will never stall the pipeline when executed after stores to BOOT or HSB.
- A doubleword store performs two memory accesses, but these will be pipelined. The last of these accesses will stall if the instruction following the doubleword is a memory access instruction other than a store to BOOT or HSB. Therefore, a non-memory instruction or another store to BOOT or HSB should be scheduled after a doubleword store to BOOT or HSB for maximum performance.

Table 8-7. Load and store instructions

Mnemonics		Operands	Description	Issue latency IRAM
ld.ub	C	Rd, Rp++	Load unsigned byte with post-increment.	2
	C	Rd, --Rp	Load unsigned byte with pre-decrement.	2
	C	Rd, Rp[disp]	Load unsigned byte with displacement.	1
	E	Rd, Rp[disp]		1
	E	Rd, Rb[Ri<<sa]	Indexed Load unsigned byte.	1
ld.ub{cond4}	E	Rd, Rp[disp]	Load unsigned byte with displacement if condition satisfied. CPU revision 2 and higher only.	1
ld.sb	E	Rd, Rp[disp]	Load signed byte with displacement.	1
	E	Rd, Rb[Ri<<sa]	Indexed Load signed byte.	1
ld.sb{cond4}	E	Rd, Rp[disp]	Load signed byte with displacement if condition satisfied. CPU revision 2 and higher only.	1

Table 8-7. Load and store instructions

ld.uh	C	Rd, Rp++	Load unsigned halfword with post-increment.	2
	C	Rd, --Rp	Load unsigned halfword with pre-decrement.	2
	C	Rd, Rp[disp]	Load unsigned halfword with displacement.	1
	E	Rd, Rp[disp]		1
	E	Rd, Rb[Ri<<sa]	Indexed Load unsigned halfword.	1
ld.uh{cond4}	E	Rd, Rp[disp]	Load unsigned halfword with displacement if condition satisfied. CPU revision 2 and higher only.	1
ld.sh	C	Rd, Rp++	Load signed halfword with post-increment.	2
	C	Rd, --Rp	Load signed halfword with pre-decrement.	2
	C	Rd, Rp[disp]	Load signed halfword with displacement.	1
	E	Rd, Rp[disp]		1
	E	Rd, Rb[Ri<<sa]	Indexed Load signed halfword.	1
ld.sh{cond4}	E	Rd, Rp[disp]	Load signed halfword with displacement if condition satisfied. CPU revision 2 and higher only.	1
ld.w	C	Rd, Rp++	Load word with post-increment.	2
	C	Rd, --Rp	Load word with pre-decrement.	2
	C	Rd, Rp[disp]	Load word with displacement.	1
	E	Rd, Rp[disp]		1
	E	Rd, Rb[Ri<<sa]	Indexed Load word.	1
	E	Rd, Rp [Ri<part> << 2]	Load word with extracted index.	1
ld.w{cond4}	E	Rd, Rp[disp]	Load word with displacement if condition satisfied. CPU revision 2 and higher only.	1
ld.d	C	Rd, Rp++	Load doubleword with post-increment.	3
	C	Rd, --Rp	Load doubleword with pre-decrement.	3
	C	Rd, Rp	Load doubleword.	2
	E	Rd, Rp[disp]	Load double with displacement.	2
	E	Rd, Rb[Ri<<sa]	Indexed Load double.	2
ldins.b	E	Rd<part>, Rp[disp]	Load byte with displacement and insert at specified byte location in Rd.	1
ldins.h	E	Rd<part>, Rp[disp]	Load halfword with displacement and insert at specified halfword location in Rd.	1
ldswp.sh	E	Rd, Rp[disp]	Load halfword with displacement, swap bytes and sign-extend.	1
ldswp.uh	E		Load halfword with displacement, swap bytes and zero-extend.	1
ldswp.w	E		Load word with displacement and swap bytes.	1
lddpc	C	Rd, PC[disp]	Load with displacement from PC.	1

Table 8-7. Load and store instructions

lddsp	C	Rd, SP[disp]	Load with displacement from SP.	1
st.b	C	Rp++, Rs	Store with post-increment.	1
	C	--Rp, Rs	Store with pre-decrement.	1
	C	Rp[disp], Rs	Store byte with displacement.	1
	E	Rp[disp], Rs		1
	E	Rb[Ri<<sa], Rs	Indexed Store byte.	2
st.b{cond4}	E	Rp[disp], Rs	Store byte with displacement if condition satisfied. CPU revision 2 and higher only.	1
st.d	C	Rp++, Rs	Store with post-increment.	2
	C	--Rp, Rs	Store with pre-decrement.	2
	C	Rp, Rs	Store doubleword.	2
	E	Rp[disp], Rs	Store double with displacement.	2
	E	Rb[Ri<<sa], Rs	Indexed Store double.	3
st.h	C	Rp++, Rs	Store with post-increment.	1
	C	--Rp, Rs	Store with pre-decrement.	1
	C	Rp[disp], Rs	Store halfword with displacement.	1
	E	Rp[disp], Rs		1
	E	Rb[Ri<<sa], Rs	Indexed Store halfword.	2
st.h{cond4}	E	Rp[disp], Rs	Store halfword with displacement if condition satisfied. CPU revision 2 and higher only.	1
st.w	C	Rp++, Rs	Store with post-increment.	1
	C	--Rp, Rs	Store with pre-decrement.	1
	C	Rp[disp], Rs	Store word with displacement.	1
	E	Rp[disp], Rs		1
	E	Rb[Ri<<sa], Rs	Indexed Store word.	2
st.w{cond4}	E	Rp[disp], Rs	Store word with displacement if condition satisfied. CPU revision 2 and higher only.	1
stcond	E	Rp[disp], Rs	Conditional store with displacement.	2/3
stdsp	C	SP[disp], Rs	Store with displacement from SP.	1
sthh.w	E	Rp[disp<<2], Rx, Ry	Combine halfwords to word and store with displacement	2
	E	Rb[Ri<<sa], Rx, Ry	Combine halfwords to word and store indexed	2
stswp.h	E	Rp[disp], Rs	Swap bytes and store halfword with displacement.	1
stswp.w	E		Swap bytes and store word with displacement.	1

8.11 Multiple data memory access instructions

These instructions perform multiple data accesses. The incremental accesses are performed as word accesses. The number of cycles is dependent on the number of registers to load or store, *n*. The issue latency must be modified as follows:

- LDM and POPM will use an additional cycle if testing of R12 is performed
- LDM and POPM that updates PC will cause a change-of-flow, which is performed in parallel with the pointer writeback and therefore has a penalty of only one cycle.
- The issue latency for HSB accesses increases if the HSB bus is busy or the slave inserts wait states.

The instructions in this group will be aborted immediately if any interrupts are pending, in order to limit the interrupt latency.

Table 8-8. Multiple data memory accesses

Mnemonics		Operands	Description	Issue latency IRAM	Issue latency HSB
ldm	E	Rp, Reglist16	Load multiple registers. R12 is tested if PC is loaded. If PC is in the register list, p=1, otherwise p=0.	1+n+p	1+2n+p
ldm	E	Rp++, Reglist16	Load multiple registers. R12 is tested if PC is loaded.	2+n	1+2n
ldmts	E	Rp, Reglist16	Load multiple registers for task switch.	1+n	1+2n
ldmts	E	Rp++, Reglist16	Load multiple registers for task switch.	2+n	1+2n
popm	C	Reglist8	Pop multiple registers from stack. R12 is tested if PC is popped.	2+n	1+2n
pushm	C	Reglist8	Push multiple registers to stack.	2+n	3+n
stm	E	Rp, Reglist16	Store multiple registers.	1+n	2+n
stm	E	--Rp, Reglist16	Store multiple registers.	2+n	3+n
stmts	E	Rp, Reglist16	Store multiple registers for task switch.	1+n	2+n
stmts	E	--Rp, Reglist16	Store multiple registers for task switch.	2+n	3+n

8.12 Branch instructions

The branch instructions cause a pipeline flush and change-of-flow if taken. Two cycles must be added to the issue latency if the branch is taken.

Table 8-9. Branch instructions

Mnemonics		Operands	Description	Issue latency
br{cond3}	C	disp	Branch if condition satisfied.	1
br{cond4}	E	disp		1
rjmp	C	disp	Relative jump.	1
ret{cond4}	C	Rs	Conditional return from subroutine with move and test of return value.	1

8.13 Call instructions

Call instructions behave similarly to branches, except that the link register (LR) must be updated. The issue latency presented in the table includes the branch penalty.

The *breakpoint* instruction takes a single cycle if Debug mode is disabled, in this case it executes as a *nop*. The breakpoint instruction updates RAR_DBG instead of LR.

Table 8-10. Call instructions

Mnemonics		Operands	Description	Issue latency
acall	C	disp	Application call	4
icall	C	Rd	Register indirect call.	4
mcall	E	Rp[disp]	Memory call.	4
rcall	C	disp	Relative call.	4
	E	disp		4
scall	C		Supervisor call	6
sscall	C		Secure State call. CPU revision 3 and higher only.	5
breakpoint	C		Breakpoint.	3

8.14 Return from execution mode instructions

The *rete* and *rets* instruction may pop the status register and return address from the system stack, and perform a branch to the return address. The *retd* instruction gets the return address and return status registers from the RAR_DBG and RSR_DBG system registers. The issue latency presented in the table includes the branch penalty.

The *rete* instruction has a latency of 12 cycles when returning from INT0-INT3 modes, 5 cycles otherwise. The *rete* instruction can be aborted by a pending interrupt.

Table 8-11. Return from execution mode instructions

Mnemonics		Operands	Description	Issue latency
retd	C		Return from debug mode	3
rete	C		Return from exception	5 / 12
rets	C		Return from supervisor call	5
retss	C		Return from Secure State call. CPU revision 3 and higher only.	5

8.15 Swap instructions

The *swap* instruction performs two atomical memory accesses, first one read and then one write.

Table 8-12. Swap instructions

Mnemonics		Operands	Description	Issue latency
xchg	E	Rd, Rx, Ry	Exchange register and memory.	2

8.16 System register instructions

This group moves data to and from the system registers. Accesses to system registers are performed in the EX stage, taking one cycle.

MTSR to SREG takes 3 cycles, MTSR to all other system registers takes 1 cycle.

Table 8-13. System register instructions

Mnemonics		Operands	Description	Issue latency
mfdr	E	Rd, SysRegNo	Move debug register to Rd.	1
mfsr	E	Rd, SysRegNo	Move system register to Rd.	1
mtdr	E	SysRegNo, Rs	Move Rs to debug register.	1
mtsr	E	SysRegNo, Rs	Move Rs to system register.	1/3
musfr	C	Rs	Move Rs to status register.	1
mustr	C	Rd	Move status register to Rd.	1

8.17 System control instructions

This group contains simple single-cycle instructions that control the behaviour of different parts of the system. The *frs*, *pref* and *sync* instructions are executed as NOP in AVR32UC.

Table 8-14. System control instructions

Mnemonics		Operands	Description	Issue latency
frs	C		Invalidate the return address stack.	1

Table 8-14. System control instructions

pref	E	Rp[disp]	Prefetch cache line.	1
sleep	E	Op8	Enter SLEEP mode.	1
sync	E	Op8	Flush write buffer.	1

8.18 Read-modify-write instructions

This group contains instructions that perform atomical bit-operations on memory addresses. These instructions require multiple cycles inside the memory controller, but these can be performed in parallel with subsequent instructions if the following instructions are not memory access instructions.

A RMW instruction performed on an address in the IRAM section executes in a single cycle if the IRAM write buffer is empty. If the write buffer is not empty, two cycles are required. The programmer can make sure the buffer is empty by ensuring that the instruction immediately before the RMW instruction is not a store or another RMW instruction.

If the RMW instruction is performed on an address in the HSB section, four cycles are needed for the RMW instruction to be executed. Therefore, if another instruction attempts to access memory within one of the three following clock cycles, up to three stall cycles will be inserted. If a memory access instruction is scheduled less than 3 cycles after the RMW to HSB instruction, 3-n stall cycles are inserted. Here n is the number of cycles used by instructions between the RMW instruction and the first memory access instruction. RMW operations to the HSB section will take additional cycles if the HSB inserts wait states.

When using RMW instructions, try to schedule code so that stall cycles are avoided.

Table 8-15. Read-modify-write instructions to IRAM section

Mnemonics		Operands	Description	Execution cycles IRAM	Execution cycles HSB
memc	E	imm, bp	Clear bit in memory.	1/2	4
mems	E	imm, bp	Set bit in memory.	1/2	4
memt	E	imm, bp	Toggle bit in memory.	1/2	4

8.19 Code example

8.19.1 Assumptions

In the example code given in this chapter, the following assumptions are made:

- r0 points to an address in the IRAM space. IRAM is an alias for r0.
- r1 points to an address in the HSB or BOOT space. HSB is an alias for r1.
- All memories and buses have 0 wait state access.
- The CPU is in a priviliged mode, so that no privilege violations occur

- All instructions are executed in the precise sequence shown below

Instruction	Cycles	Description
add r5, r0	1	
sub r5, r5	1	
ssrf AVR32_SREG_C	1	
ssrf AVR32_SREG_GM	3	SSRF to bits 31-16 takes 3 cycles
max r6, r1, r0	1	
mul r5, r1	1	
mac r5, r1	1	1 cycle since r5 is already in the accumulator cache
mac r3, r1	2	2 cycles since r3 is not in the accumulator cache
mac r3, r1	1	1 cycle since r3 is not in the accumulator cache
macs.d r2, r1, r2	4	4 cycles since register pair r3:r2 is not in the accumulator cache
mulwh.d r6, r5, r1:t	1	48 bit result calculated and written back in 1 cycle
st.w IRAM[0], r6	1	
divs r4, r5, r6	35	
ld.w r8, IRAM++	2	Load with postincrement takes two cycles
satadd.w r4, r8, r9	1	
ld.w r4, IRAM[4]	1	
add r4, r4	1	No data hazard after loads from IRAM
ld.w r5, IRAM[8]	1	
ld.w r6, IRAM[12]	1	Loads from IRAM can be adjacent without any stalling
ldm IRAM++, r5, r6, r7, r8	5	ldm from IRAM takes 1+n= 5 cycles when loading 4 registers
mfsr r8, AVR32_SREG	1	
cbr r8, 0	1	
mtsr AVR32_SREG, r8	3	mtsr to SREG takes 3 cycles, 1 cycle required for other sysregs
st.w IRAM[0], r5	1	
st.w IRAM[4], r6	1	Stores to IRAM can be adjacent without any stalling
nop	1	nop takes 1 cycle
ld.w r5, HSB[0]	2	Reading from memories on the bus takes 2 cycles
ld.w r6, HSB[4]	2	HSB bus reads are not pipelined, each read takes 2 cycles
st.w HSB[8], r6	1	HSB store done in background if 2 next insn is not mem access
add r5, r6	1	Nonmem insn scheduled after HSB store to avoid stall
and r7, r8	1	Nonmem insn scheduled after HSB store to avoid stall
st.w HSB[8], r6	2	First of consecutive HSB stores requires extra cycle to start
st.w HSB[12], r7	1	Consecutive HSB stores are pipelined.
st.w HSB[16], r8	1	Consecutive HSB stores are pipelined.
add r5, r6	1	Consecutive HSB stores followed by 1 nonmem insn do not stall

Instruction	Cycles	Description
ldm HSB++, r5, r6, r7, r8	9	1+n=1+2r where r=#regs when reading from HSB addresses
stm HSB, r5, r6, r7, r8	6	1+n=1+(1+r) where r=#regs when writing to HSB addresses
add r6, r5	1	Instruction not using the write buffer
memc IRAM, 3	1	Requires only one cycle because write buffer was empty
st.w IRAM[4], r8	1	Instruction filling the write buffer
memc IRAM, 3	2	Requires 2 cycles because write buffer was not empty
mul r5, r9	1	
memc HSB, 7	4	Next instruction has a memory access, 3 stall cycles needed
ld.w r4, IRAM[16]	1	
memc HSB, 7	1	Memory not accessed in the 3 following clock cycles, no stall
sub r7, r4	1	
mul r6, r9	1	
or r5, r8	1	

9. OCD system

9.1 Overview

The AVR32 CPU is targeted at a wide range of 32-bit applications. The CPU can be delivered in very different implementations in various ASIC's, ASSP's, and standard parts to satisfy requirements for low-cost as well as high-speed markets. According to the cost sensitivity and complexity of these applications, a similar span in debug complexity must be expected. While some users expect very simple debug features, or none at all, others will demand full-speed trace and RTOS debug support. This also applies to the debug tools: While the simplest development takes place on simulators and development boards, most will require basic on-chip debug emulators, and a few will require complex emulators with full-speed trace.

To match these criteria, the AVR32 OCD system is designed in accordance with the Nexus 2.0 standard (IEEE-ISTO 5001™-2003), which is a highly flexible and powerful open on-chip debug standard for 32-bit microcontrollers.

9.1.1 Features

- Nexus compliant debug solution
- OCD supports any CPU speed
- Execute debug specific CPU instructions (debug code) from program memory monitor or external debugger
- Debug code can read and write all registers and data memory
- Debug code can communicate with debugger through the debug port
- Debug mode can be entered by external command, breakpoint instruction, or hardware breakpoints
- Six program counter hardware breakpoints are supported
- Two data breakpoints are supported
- Breakpoints can be configured as watchpoints (flagged to the external debugger)
- Hardware breakpoints can be combined to give break on ranges
- Real-time program counter branch tracing
- Real-time data trace
- Real-time process trace
- Nexus Class 2+

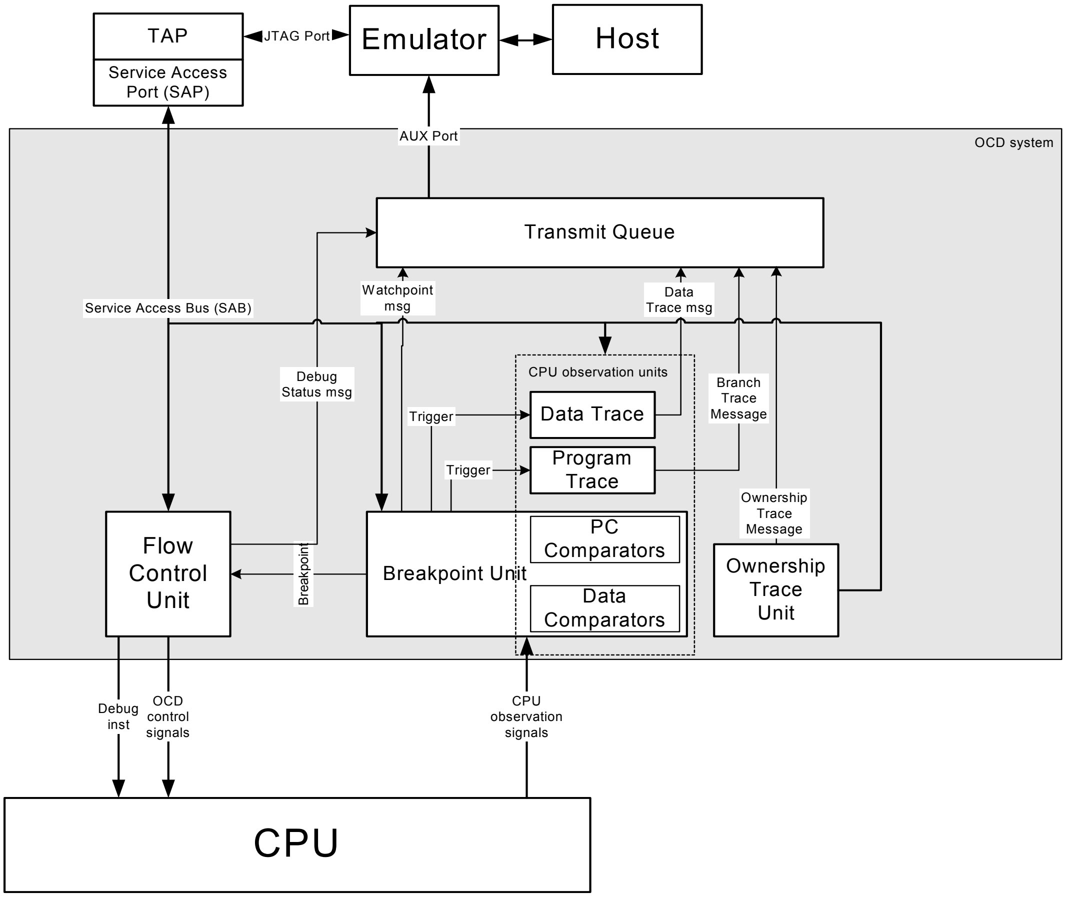
9.1.2 OCD controller overview

The OCD system interfaces provides the external debugger with access to the on-chip debug logic through the JTAG port and the Auxiliary (AUX) port, as shown in Figure 9-1. The operation is described briefly below and in more detail in separate chapters.

9.1.2.1 Host, debugger, and emulator

At the host side, the user debugs his software using a source level debugger, which can read his compiled and linked object code. The source level debugger accesses features in the emulator and OCD system through an API (defined by the vendor or based on the Nexus recommendations), which constitutes the abstract interface between the source level debugger and the emulator. The API translates high-level functions, such as setting breakpoints or reading memory areas, to sets of low level commands understood by the OCD controller. Certain operations (such as reading the register file) may require running sections of debug code on the CPU, which can also be handled in this level. The emulator translates the communication from the host into commands transmitted to the target over the JTAG port. If trace is enabled, trace messages are transmitted from the device on the Nexus-defined auxiliary (AUX) port. The AUX port can be scaled to the number of output pins needed to sustain the estimated bandwidth requirement. The Nexus protocol defines the format of the messages and signals, the pin count options and pinout of the debug port, and the type of connector used.

Figure 9-1. Block diagram of the OCD system (shaded) and its main connections.



9.1.2.2 *Accessing the debug features*

A number of blocks handle the various debug functions specified by the Nexus standard. The emulator communicates with registers in these blocks by commands on the JTAG port, as specified by the Nexus standard. OCD registers are typically used for configuration, control, and status information. Trace information and debug events can also generate messages to be transmitted on the AUX port.

Registers are indexed and are accessed through Read Register and Write Register messages from the emulator. Alternatively, they can be accessed by the CPU through *mtdr* and *mfdr* instructions, which gives a debug monitor in the CPU access to most of the debug features in the OCD system, as described in "OCD Register Access" on page 98.

9.1.2.3 Transmit Queue

Trace and watchpoint messages are inserted into the Transmit Queue (TXQ) before being transmitted on the AUX port. This provides some flexibility between the peak rate of trace message generation and the average rate of message transmission on the AUX port.

9.1.2.4 Flow Control Unit

The Flow Control Unit (FCU) can bring the CPU into and out of Debug Mode, and control the CPU operation in Debug Mode. The behavior is controlled by accessing OCD registers.

Debug Mode can be configured as OCD Mode or Monitor Mode. In OCD mode, The CPU fetches instructions from the Debug Instruction Register. If the register is empty, the CPU is halted. In Monitor Mode, the CPU fetches debug instructions from a monitor code in the program memory, and the Debug Instruction Register is not used.

The FCU also handles single stepping by returning the CPU to normal mode, letting the CPU fetch one instruction from the program memory, and then returning to Debug Mode on the following instruction.

9.1.2.5 Breakpoint modules

A number of instruction and data breakpoint modules can be configured for run-time monitoring of the instruction fetches and data accesses by the CPU. The modules can report if the monitored operation matches a predefined address, alternatively, also a data value. The modules operate on virtual addresses.

A breakpoint will bring the CPU into Debug Mode. Watchpoints are reported to the debugger, but does not affect CPU operation. A watchpoint can also be configured to start or stop data and program trace.

The breakpoint modules can be combined to produce a watchpoint or breakpoint. Complex breakpoint/watchpoint conditions are supported, e.g. trigger when a specific procedure writes a certain variable with a specific value.

9.1.2.6 Program and Data Trace

The Program Trace Unit sends Branch Trace Messages to the debugger, which allows the program flow to be reconstructed. To keep the amount of debug information low to save bandwidth, only change of program flow are reported (such as unconditional branches, taken conditional branches interrupts, exceptions, return operations, and load operations with PC as destination), hence the term "branch tracing". Messages are typically relative to the previously transmitted message, to be able to compress information as much as possible. Thus, the trace messages are sent out in temporal order, and regularly, synchronization messages with uncompressed, absolute addresses, are transmitted in case synchronization is lost.

The Data Trace Unit similarly traces data accesses, for read or write accesses, or both. Similar relative address compression and synchronization schemes are used for Data Trace Messages. Since new trace messages can be generated before the previous ones have been transmitted, all trace messages are queued before being transmitted by the AUX interface. If the queue overflows, the CPU can be halted to avoid losing trace information, or an error message followed by synchronization trace messages will be transmitted.

9.1.2.7 OS debug support

Applications developed on an OS platform places special requirements on the OCD controller and the debug software. For high-level debugging, the user will want to see which process is running at any time, without having to interrupt the CPU or trace the program flow. This is accomplished through Ownership Trace Messaging, in which the process ID of the running process is reported at every process switch. The CPU writes the process ID to an OCD register in the Ownership Trace Unit, which in turn generates an Ownership Trace Message.

9.1.2.8 Timestamps

The emulator can tag events with a timestamp when they are extracted from the OCD system and transmitted to the emulator, to provide timing information for these events when they are transmitted to the debug host. However, due to the delay of the transmit queue and transmit time over the AUX port, this timing will have limited accuracy. To compensate for this, the $\overline{\text{EVTO}}$ pin can be configured to toggle every time a message is inserted into the Transmit Queue, thus indicating very precisely when each event occurs. The emulator would then store a queue of timestamp tags with each event, and associate each tag with the corresponding message, as they are extracted on the AUX port.

9.2 CPU Development Support

The OCD system can bring CPU into and out of Debug Mode, and control the CPU operation in Debug Mode. The behavior is controlled by OCD register configuration, stop commands from the debugger, or breakpoints. The OCD registers can be accessed by Nexus messages or from the CPU as memory-mapped registers.

9.2.1 Debug Mode

Debug Mode is an execution mode dedicated to application debugging and is not intended for running application code. Debug Mode can execute a debug code either from an external debugger through the OCD system (OCD Mode), or from a debug routine in program memory (Monitor Mode). The debug code will typically read out system registers and information about the various processes running in the system before restarting.

The Nexus class 2+ compliant OCD system contains breakpoint and trace modules, and other features for debugging code on the CPU. These features are generally accessible both in OCD Mode and Monitor Mode. In OCD Mode, the debugger accesses the features through messages over the AUX debug port, and in Monitor Mode, the CPU accesses the features through *mtdr* and *mfdr* instructions. The OCD system runs at system speed to stay synchronous with the CPU at all times. If the CPU is in a low-power sleep mode, it is woken up before entering Debug Mode.

9.2.1.1 Operations in Debug Mode

Debug Mode is characterized by the Debug (D) bit in the Status Register (SR) in the CPU. Debug Mode is a privileged mode, and all legal instructions and memory operations are permitted Illegal opcodes or memory operations which would normally cause an exception will be ignored in Debug Mode.

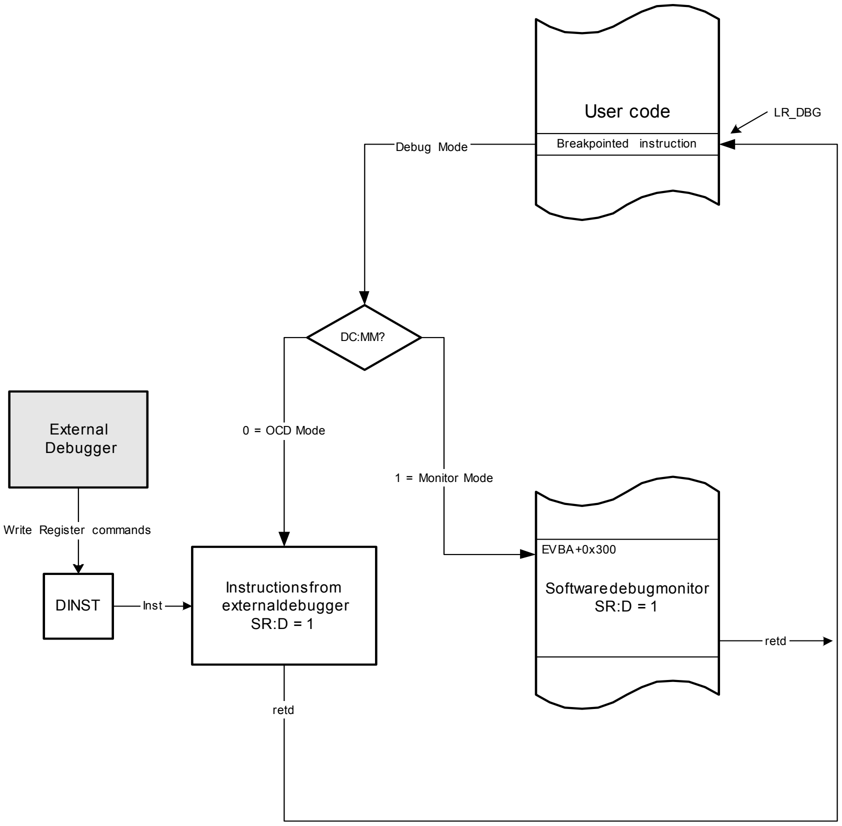
The Debug Mode has a dedicated Link and Return Status Register (RAR_DBG and RSR_DBG, respectively) but no other masked registers. RAR_DBG and RSR_DBG are not observable as part of the register file, only as system registers. The register file view is mapped according to the mode bits in the Status Register (M[2:0]). These bits are set to the exception context when entering Debug Mode, but can be changed freely within Debug Mode by writing to SR. In this way, different register contexts can be observed and modified, while maintaining the execution and access privileges of Debug Mode.

Debug Mode is exited by the *retd* instruction, both in Monitor Mode and OCD Mode. This restores PC from RAR_DBG and SR from RSR_DBG.

9.2.1.2 *A typical debug session flow*

Figure 9-2 shows an example of a typical flow in Debug Mode. A software or hardware breakpoint aborts the execution of an instruction and causes Debug Mode to be entered. If the Monitor Mode (MM) bit in the Development Control (DC) OCD register is set, Monitor Mode is entered, and the CPU jumps to the software debug monitor starting at EVBA+0x01C. Otherwise, OCD Mode is entered, and the CPU stalls while waiting for instructions to be entered by the external debugger through the Debug Instruction (DINST) OCD register. In either case, the D bit in the CPU Status Register is set during the whole debug session, giving access to all privileged operations. Any number of instructions can be executed before returning to the breakpointed instruction by the *retd* instruction. RAR_DBG stores the address of the breakpointed instruction, and manipulating RAR_DBG in Debug Mode is useful if a different return address is desired (for instance, to avoid repeated hits on a *breakpoint* instruction).

Figure 9-2. Example of flow in Debug Mode.



9.2.2 Monitor Mode

If the Monitor Mode (MM) bit in the Development Control register (DC) is set, the CPU will enter Debug Mode in Monitor Mode. Instructions are fetched from the monitor code located in the program memory at the Exception Vector Base Address (EVBA) + 0x01C. The monitor code contains the necessary mechanisms to read and modify CPU and system registers, and memory areas. All other exceptions and interrupts are masked by default when entering Monitor Mode, but the monitor code can explicitly unmask interrupts to allow critical interrupts to be serviced while the system is being debugged.

The monitor code will typically communicate with an external debug tool, or (in cases of advanced systems like PDA's) a debug tool running within the application (self-hosted debugger). Communication with the external tool may take place over any communication link present in that device (e.g. USB, RS232), if such a communication line can be reserved for debug purposes.

Alternatively, the Debug Communication Mechanism in the OCD system can be used to communicate between the CPU and emulator over the JTAG port. This is a set of OCD registers which can be written by the CPU or emulator, allowing a communication protocol to be developed in software. This mechanism can be used in any privileged CPU mode, including OCD Mode.

Monitor Mode is exited with the *retd* instruction.

9.2.2.1 Debugging a monitor code

Each execution mode has a mask bit in SR, which indicates if a request to enter that mode will be taken or masked. The default priority of modes are reflected in these bits: When entering an execution mode, modes of the same or lower priority are masked. Privileged modes can override the mask, to dynamically change priorities (e.g. to allow critical interrupts to be serviced).

By default, Debug Mode has priority above all other execution modes. This implies that any supervisor or user code can be interrupted by Debug Mode. Other modes can be explicitly unmasked by a monitor code to allow critical interrupts to be serviced. By default, Debug Mode is masked by the Debug Mask (DM) bit in SR when executing in Monitor Mode. The Monitor Mode can stack away the RAR_DBG and RSR_DBG and then explicitly clear the DM bit to enable Debug Mode to be re-entered. If a debug exception occurs in Monitor Mode, the OCD system will bring the CPU into OCD Mode, even if the MM bit is set. This allows Monitor Mode programs to be debugged.

9.2.3 OCD Mode

If the Monitor Mode (MM) bit in the Development Control register (DC) is cleared, the CPU will enter Debug Mode in OCD Mode. When the CPU is in OCD Mode, the Debug Status (DBS) bit in the Development Status (DS) register is set, in addition to the D bit in SR in the CPU. OCD Mode is similar to Monitor Mode, except that instructions are fetched from the OCD system. OCD instructions are loaded by the debug tool by writing the opcode to the Debug Instruction register (DINST). Once an instruction is written to DINST, the CPU will fetch it, and the Instruction Complete bit in DS (DS:INC) will be cleared until the CPU has completed the operation. The CPU is then halted until DINST is written again.

The first instruction entered must be aligned to the MSB of DINST. A sequence of instructions can be entered to DINST one word at a time, in the same sequence they would appear in program memory, i.e. they do not need to be word aligned. If the upper halfword of an extended instruction is written to the lower halfword of DINST, the lower halfword of the instruction is written as the upper halfword of DINST in the next access. If the last instruction in a sequence is written to the upper halfword of DINST, the lower halfword should be written with a nop opcode.

See Figure 9-3 for an illustration of a sequence of operations used to execute instructions in OCD Mode.

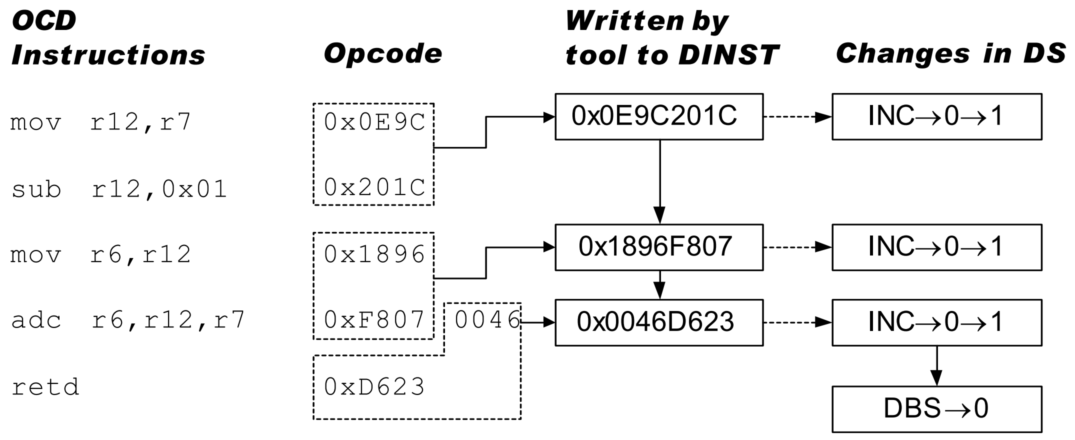
Any instruction valid in Monitor Mode is also valid in OCD Mode. Memory operations can be conducted without any special synchronization with external hardware.

All OCD units can be configured while the CPU executes in OCD Mode, but the following debug features are disabled:

- PC breakpoints
- Data breakpoints
- Watchpoints
- Program Trace
- Data Trace

OCD Mode is exited by writing the *retd* instruction to DINST.

Figure 9-3. Executing instructions on the CPU in OCD Mode.



9.2.4 Entry into Debug Mode

Debug Mode can only be entered when the OCD is enabled, and Debug Mode is not masked. The following ways of entry are then possible:

- Debug request from the debugger
- Program counter breakpoint
- Data address or value breakpoint
- *breakpoint* instruction
- Trapping opcode 0x0000
- Single step
- Event on $\overline{\text{EVTI}}$ pin
- Abort command from the debugger

The debugger can identify the condition which caused entry into Debug Mode by examining the status bits in the Development Status register (DS). Each cause of entry has a particular bit associated with it. Several exceptions can trigger simultaneously, causing more than one bit to be set.

Note that any privileged CPU mode may write the SR:D bit to one directly, but this will not cause entry to Debug Mode.

9.2.4.1 *Debug request*

The debugger may want to stop CPU operation, unrelated to current instruction execution, e.g. if the user presses a "STOP" button in the debug tool GUI. The debugger will then write the Debug Request (DBR) bit in the Development Control Register (DC). This causes the CPU to enter Debug Mode on the next instruction to be executed, before execution.

9.2.4.2 *Program counter breakpoint*

The Program Counter breakpoints can be configured to halt the CPU when executing code at a specific address, or address range. This will cause the CPU to be halted before the breakpointed instruction is executed.

The Ignore First Match (IFM) bit in the Development Control (DC) register should be written to one before exiting Debug Mode, to avoid re-triggering the program breakpoint. This bit only prevents program breakpoints from re-triggering. If the instruction causes a breakpoint for another reason (e.g. a *breakpoint* instruction or a data breakpoint), Debug Mode will be re-entered.

9.2.4.3 *Data address or value breakpoint*

CPU memory accesses can be monitored by data breakpoint comparators in the OCD system. If the access matches a set of predefined conditions (e.g. address, value, or access type), Debug Mode is entered after the memory operation completes, but before the next instruction is executed.

Data breakpoints are precise, halting on the instruction immediately after the memory operation which caused the breakpoint. The CPU will return to the first non-executed instruction when a *retd* is executed.

9.2.4.4 *breakpoint instruction*

The *breakpoint* instruction is programmed along with the object code into the program memory or instruction cache, and is decoded by the CPU. When this instruction is scheduled for execution and Debug Mode is enabled, the CPU will enter Debug Mode. If Debug Mode is disabled (e.g. masked by the DM bit in the Status Register, or DBE in DC is zero), the *breakpoint* instruction will execute as a *nop* (no operation).

For devices based on volatile program memory, the *breakpoint* instruction can be dynamically inserted into the code by the debug tool, enabling an unlimited number of program breakpoints in the code. This involves replacing an existing opcode with a breakpoint instruction. The replaced opcode has to be re-inserted before exiting Debug Mode. Note that this is only possible in OCD Mode.

For devices based on non-volatile program memory, the *breakpoint* instruction can be statically compiled or linked into the code before downloading, marking all points the program can be halted. Debug Mode will be entered for all breakpoints (if Debug Mode is enabled), and the debugger would return immediately if it does not want to halt at a particular breakpoint location in the code.

The breakpoint will be taken before the *breakpoint* instruction is actually executed. This has the effect that the CPU will return from Debug Mode to the same *breakpoint* instruction, re-entering Debug Mode immediately, unless the OCD system is configured to modify the return address or replace the *breakpoint* instruction from the instruction flow. The IFM bit does not have an effect when Debug Mode returns to a *breakpoint* instruction.

9.2.4.5 *Trapping opcode 0x0000*

In Flash-based microcontrollers, the opcode 0x0000 can overwrite any other opcode without having to erase and reprogram the Flash. Therefore this instruction can enter Debug Mode, as for the *breakpoint* instruction. However, the opcode 0x0000 is also a valid part of the instruction set (ADD R0,R0 in AVR32) and can be part of the software to be debugged. Therefore, the user must write the DC:TOZ (Trap Opcode Zero) bit to one to enable this feature.

The DS:BOZ bit will be set if Debug Mode is entered due to a trapped 0x0000 instruction. The debugger must then identify whether this opcode belongs to the original object file or has been inserted by the debugger as a software breakpoint. If it was part of the object file, the debugger should use the Instruction Replacement to return to the program, and insert the 0x0000 opcode in DINST. Executing 0x0000 during Instruction Replacement only performs an ADD R0,R0 operation without re-entering Debug Mode.

9.2.4.6 *Single stepping*

The debugger will typically allow the user to step through the application source or object code, line by line. This single stepping can be either of step-into or step-over type. Step-into will execute exactly one instruction and halt the CPU at the start of the next instruction, regardless of whether this instruction is part of the main program, subroutine, interrupt, or exception. Step-over will execute the current instruction and any lower-level events generated before the following instruction (including subroutines, interrupts, and exceptions).

Step-over in the object code and all single stepping in the source code are implemented by configuring a program breakpoint on the address of the next object code instruction where the debugger expects to halt.s

Step-into is implemented in OCD hardware and is controlled by the Single Step (SS) bit in the Development Control register. When Debug Mode is exited by retd, exactly one instruction from the program memory will be executed before Debug Mode is re-entered. This mechanism works identically for OCD and Monitor Mode.

9.2.4.7 *Event on $\overline{\text{EVTI}}$ pin*

If the Event In Control (EIC) bits in DC are written to 0b01, a high-to-low transition on the $\overline{\text{EVTI}}$ pin will generate a breakpoint. $\overline{\text{EVTI}}$ must stay low for one CPU clock cycle to guarantee that the breakpoint will trigger. The External Breakpoint (EXB) bit in DS will be set when a breakpoint is entered due to an event on the $\overline{\text{EVTI}}$ pin.

9.2.4.8 *Abort command*

Some software errors could cause the CPU to get stuck in a state which does not allow Debug Mode to be entered through the mechanisms described above. An example is if a privileged mode writes SR:DM to one, without clearing the bit.

To prevent the debugger from hanging indefinitely, the debugger can write the DC:ABORT bit to one after some timeout period, and force the CPU to enter Debug Mode. The abort command behaves identical to a debug request, except that the DM bit and any pending exception will be ignored, regardless of exception priority. The RAR_DBG and RSR_DBG will reflect the last non-executed instruction, which can aid in locating the error.

If Debug Mode is entered due to an abort command, DS:DBA will be set, as for debug requests.

9.2.5 Exceptions and Debug Mode

Debug Mode has priority over any execution mode, so that breakpoints can be set in exception and interrupt routines. However, if a breakpoint is set on an instruction which triggers a critical exception, the breakpoint is flushed. Critical exceptions are exception which are asynchronous to the CPU (interrupts), exceptions which invalidate the currently fetched instruction (e.g. instruction address exceptions), and exceptions which indicate that the system has become unstable and should abort the program flow (e.g. bus error). The complete list of exceptions with higher priority than Debug Mode are listed in the exception chapter in the AVR32 Architecture Manual.

If a PC breakpoint, a breakpoint instruction, or a trapped 0x0000 opcode is flushed by an exception, Debug Mode will not be entered. If another type of breakpoint has triggered, Debug Mode will be entered on the first instruction in the exception handler.

In the rare cases where the first instruction in a critical exception also triggers a critical exception (e.g. if EVBA is set incorrectly, triggering an infinite loop of instruction address exceptions), the debugger must write the DC:ABORT bit to one to halt the CPU and enter Debug Mode to identify the error.

9.2.6 Instruction replacement

A convenient way of implementing an unlimited number of instruction breakpoints is letting the debugger replace an instruction by a *breakpoint* instruction. This mechanism is only available in OCD Mode on devices implemented with writeable program memory or writeable instruction cache. If this instruction executes, Debug Mode will be entered, and the debugger identifies the breakpointed location. When returning, the breakpoint instruction must be replaced by the original instruction. The debugger will write the Instruction Replace (IRP) bit in DC and the appropriate instruction in the Debug Instruction Register and its corresponding PC value in the Debug Program Counter (DPC). When *retd* is executed, PC and SR are restored, but one more instruction is fetched from the OCD system before returning to fetching from program memory.

Note that instruction replacement operates on word boundaries. The debugger must store the whole word containing the replaced opcode before inserting the *breakpoint* instruction. Also note that DPC should always be written when performing an instruction replacement to ensure the correct instruction is executed.

The debugger will then perform the following sequence when exiting OCD Mode. Note that RAR_DBG is accessed through executing CPU instructions through the Debug Instruction register (DINST). The same sequence can be used both for compact and extended instructions, regardless if the extended instruction is unaligned (in which case only the upper halfword of the instruction is replaced).

1. Write RAR_DBG to the Debug Program Counter.
2. Increment RAR_DBG by 2 or 4, so the register points to the start of the next word in the program memory.
3. Write 1 to Instruction Replace (IRP) in DC.
4. Write a *retd* instruction to DINST. The CPU will exit Debug Mode and stall while waiting for new instructions.
5. Write the stored word to DINST. This instruction is fetched by the CPU, and the CPU continues normal program execution.

9.2.6.1 *Instruction replacement example*

Table 9-1 shows an example of a code where the user wants to insert a breakpoint.

Table 9-1. Example of a user code section

PC value	Opcode	Instruction
0x000010	0x0E9C	mov r12,r7
0x000012	0x201C	sub r12,0x01
0x000014	0xC0AC	rcall label1
0x000016	0xF8070046	adc r6,r12,r7
0x00001A	0x2027	sub r7,0x02

The tool wants to insert a software breakpoint on the instruction "adc r6,r12,r7" on PC=0x000016. This is an extended instruction, and only the upper halfword needs to be replaced by the breakpoint instruction.

1. The upper halfword is contained within the word located at 0x000014, and the debug tool stores this value (0xC0ACF807).
2. The debugger writes a breakpoint instruction (opcode 0xD673) to location 0x000016 in the CPU's program memory to replace the most significant word of the breakpointed instruction.
3. When the breakpoint instruction executes, the CPU will enter OCD Mode, and DS:DBS and DS:SWB are set, indicating that OCD Mode is entered due to a software breakpoint.
4. The tool performs a normal sequence of operation in OCD Mode.
5. When the tool is ready to return to normal CPU operation, it reads the RAR_DBG value to find the return address.
6. The tool inserts CPU instructions to DINST to increment RAR_DBG by 2, so it is aligned to the next word in the program memory.
7. The tool inserts a "retd" instruction to DINST. The tool will receive a Debug Status message, which indicates that the CPU has exited OCD Mode, and is now waiting for one more instruction from the tool.
8. The tool writes the return address (0x000016) to the Debug Program Counter (DPC).
9. The tool looks up the stored instruction word (based on the return address) and writes this value (0xC0ACF807) to the Debug Instruction Register (DINST). The CPU now resumes normal operation.

9.2.7 Sleep Mode

If the CPU is in sleep mode, it will not receive clocks nor respond to an OCD request from the debugger. Thus, if the Debug Request bit in DC is written to one while the CPU is in sleep mode, the CPU will automatically return to active mode. The instruction following the sleep instruction will be tagged with an OCD exception, and the CPU will jump directly to Debug Mode. The normal debug procedure can be followed while executing in Debug Mode. If Debug Mode is entered from sleep mode, the Stop Status (STP) bit in the Development Status register will be set.

When returning from Debug Mode, the CPU will by default return to the instruction following the sleep instruction. The debugger can handle this situation in two ways:

Ignore the problem, effectively waking the CPU from sleep mode on a debug request.

Decrement RAR_DBG in Debug Mode to return to the sleep instruction. This places the CPU back into sleep mode after exiting Debug Mode.

9.2.8 OCD Register Access

The OCD registers control the OCD system. Their specification is based on the Nexus Recommended Registers as outlined in the Nexus Standard Specification [IEEE-ISTO 5001™-2003]. All registers can be accessed through the JTAG interface.

9.2.9 OCD features in Debug Mode

When the CPU executes in Debug Mode, certain OCD features will be disabled. The following table indicates how the various OCD features will behave in Debug Mode. For more information on the specific features, please see the indicated page.

Table 9-2. OCD features in Debug Mode

Feature	Available in Debug Mode?
Program Breakpoints (HW)	Yes, in Monitor Mode when SR:DM is cleared
Software Breakpoints	Yes, in Monitor Mode when SR:DM is cleared
Data Breakpoints	Yes, in Monitor Mode when SR:DM is cleared
Watchpoints (program and data)	Yes, in Monitor Mode
Program Trace	No
Data Trace	No
Ownership Trace	Yes
Debug Communication Mechanism	Yes

9.2.10 OCD Registers Accessed by CPU

A monitor program running on the target can access the OCD registers through *mtdr* and *mfdr* instructions. These instructions transfer data between a register in the register file and an OCD register, according to the register index given in "OCD Register Summary" on page 153. These instructions can also be used in OCD mode to transfer information from the register file and system registers to the debugger, through the Debug Communication Mechanism.

9.2.11 Runtime write access to OCD registers

The OCD registers can always be accessed by JTAG when the when the OCD system is not enabled or the CPU is in OCD Mode. The OCD registers can also be read by JTAG at any time, and by the CPU in any privileged mode.

When the CPU is in other modes - either running normal code, or executing in Monitor Mode - the OCD registers can be written by JTAG as specified in Table 9-3. If the registers are accessed in another way than specified, undefined operation may result.

The OCD Register Protect (ORP) bit in DC define the allowed write access to OCD registers in privileged modes. If the ORP bit in DC does not allow CPU access to OCD registers in the currently executing mode, only PID and DCCPU can be written. Illegal access to the registers will be ignored with no error reporting.

Table 9-3. OCD Register access

Register	Can be written by JTAG while CPU is running?	Can be written by CPU in Monitor Mode?
Development Control (DC)	Yes	Yes
Watchpoint Trigger (WT)	Yes	Yes
Data Trace Control (DTC)	Can be written to disable / enable trace channels.	Yes
Data Trace Start Address (DTSA) Channel 1 to 2	Can only be written while trace channel is disabled	Yes
Data Trace End Address (DTEA) Channel 1 to 2	Can only be written while trace channel is disabled	Yes
PC Breakpoint/Watchpoint Control (BWC)	Can be written to disable / enable watchpoints / breakpoints.	Yes, if SR:DM is set.
Data Breakpoint/Watchpoint Control (BWC)	Can be written to disable / enable watchpoints / breakpoints.	Yes, if SR:DM is set.
PC Breakpoint/Watchpoint Address (BWA)	Can only be written while breakpoint / watchpoint is disabled	Yes, if SR:DM is set or breakpoint disabled.
Data Breakpoint/Watchpoint Address (BWA)	Can only be written while breakpoint / watchpoint is disabled	Yes, if SR:DM is set or breakpoint disabled.
Breakpoint/Watchpoint Data (BWD)	Can only be written while breakpoint / watchpoint is disabled	Yes, if SR:DM is set or breakpoint disabled.
Ownership Trace Process ID (PID)	Yes	Yes
Debug Instruction Register	No	No
Debug Program Counter	No	No
Debug Communication CPU (DCCPU)	Yes	Yes
Debug Communication Emulator (DCEMU)	Yes	Yes

9.2.12 OCD Interrupts

To support custom debug protocols running in software the OCD system support giving interrupts to the CPU when DCEMU is written or when DCCPU is read. A software protocol handler can then be triggered by these interrupts instead of having to poll DCSR to see if the data in DCCPU or DCEMU has been read or written.

To enable these interrupts the user must do the following:

- Program the interrupt controller with the correct priority and handler address for the interrupt.
- Enable the interrupts from the OCD by setting the corresponding bits in DCCR

- Turn off the interrupt masks in the CPU

When an interrupt occurd the CPU will jump to the interrupt handler routine and process the interrupt. The interrupt handler must clear the interrupt before leaving this routing. This is done by witing a zero to DCEMUDI or DCCPURI for DCEMU reads and DCCPU writes respectively:

9.2.13 Messages

9.2.13.1 Debug Status (DEBS)

This message is output when the CPU enters or exits Debug Mode or a low-power mode. The message is output whenever the AUX port is enabled. The STATUS field of this message contains the information in the Development Status register. The field will contain these values:

- The CPU enters Debug Mode: STATUS bits indicate cause of entry to Debug Mode. DBS is set if OCD Mode was entered.
- The CPU exits Debug Mode: STATUS = 0. This includes exiting Debug Mode by writing DC:RES.
- The CPU enters a low-power mode: Only the STP bit is set, while the other bits are zero.
- The CPU exits a low-power mode: STATUS = 0

Table 9-4. Debug Status

Debug Status Message			
Packet Size	**Packet Name**	**Packet Type**	**Description**
32	STATUS	Fixed	The contents of the Development Status register.
6	TCODE	Fixed	Value = 0

9.2.14 Registers

9.2.14.1 Device ID Register (DID)

The Device ID Register (DID) provides key attributes to the development tool concerning the embedded processor. This is the same as the value returned by the JTAG ID instruction.

Table 9-5. DID Register

R/W	**Bit Number**	**Field Name**	**Init. Val.**	**Description**
R	31:28	RN	Part specific	RN - Revision Number
R	27:12	PN	Part specific	PN - Product Number
R	11:1	MID	0x01F	Manufacturer ID 0x01F = ATMEL
R	0	Reserved	1	Reserved This bit always reads as 1

9.2.14.2 Nexus Configuration Register (NXCFG)

The Nexus Configuration Register (NXCFG) provides key information about the specific implementation of the CPU and OCD architecture, and the configuration of the Nexus development features on this device. This information is static, and may be used to develop generic Nexus debuggers which will work across a family of AVR32 devices with different Nexus configurations.

Table 9-6. Nexus Configuration Register

R/W	Bit Number	Field Name	Init. Val.	Description
R	31:29	Reserved	0	
R	28	NXDMA	0	Direct Memory Access support 0 = Not supported 1 = Supported
R	27:25	NXDTC	0	Data Trace Channels 0 = Not supported 1 = Supported
R	24	NXDRT	0	Data Read Trace Support 0 = Not supported 1 = Supported
R	23	NXDWT	0	Data Write Trace Support 0 = Not supported 1 = Supported
R	22	NXOT	0	Ownership Trace support 0 = Not supported 1 = Supported
R	21	NXPT	0	Program Trace support 0 = Not supported 1 = Supported
R	20:17	NXMDO	6	AUX MDO pins 0 = no MDO or $\overline{\text{MSEO}}$ pins n = n MDO pins, NXMSEO $\overline{\text{MSEO}}$ pins
R	16	NXMSEO	1	AUX MSEO pins 0 = 1 $\overline{\text{MSEO}}$ pin 1 = 2 $\overline{\text{MSEO}}$ pins
R	15:12	NXDB	2	Number of Data breakpoints
R	11:8	NXPCB	6	Number of PC breakpoints
R	7:4	NXOCD	0	OCD Version 0000 = AVR32AP OCD 0001 = AVR32UC OCD Other = Reserved
R	3:0	NXARCH	0	Architecture 0000 = AVR32B 0001 = AVR32A Other = reserved

9.2.14.3 Debug Communication CPU Register (DCCPU)

If the CPU wants to transmit data to the debugger tool, it writes data to the Debug Communication CPU Register using mtdr. By writing this register, a dirty bit is set in the Debug

Communication Status Register. The emulator should poll the status register and read DCCPU if the dirty bit is set.

Table 9-7. Debug Communication CPU Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	31:0	DATA	0x0000_ 0000	Data Value Data written by CPU

9.2.14.4 *Debug Communication Emulator Register (DCEMU)*

When the emulator writes to this register, a dirty bit is set in the Debug Communication Status register. The CPU can poll this bit to see if DCEMU contains unread data..

Table 9-8. Debug Communication Emulator Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	31:0	DATA	0x0000_ 0000	Data Value Data written by Emulator

9.2.14.5 *Debug Communication Status Register (DCSR)*

To avoid overruns the CPU must poll this register before writing a new value to DCCPU. Note that the bits in this register are not automatically cleared in OCD mode. This allows a debugger to update views and observe the system without accidentally modifying the DCSR register.

The OCD system can produce interrupts when the DCEMU register has been updated and when the DCCPU register is read. The CPURI and EMUDI flags are set on the interrupt events, but are cleared by software by writing the DCSR register. To enable the interrupts the corresponding bits in the DCCR register has to be set and the Interrpt controller has to be programmed.

Table 9-9. Debug Communication Status Register

R/W	Bit Number	Field Name	Init. Val.	Description
R	31:4	Reserved	0x0000_ 0000	Reserved These bits are reserved, and will always read as 0
R/W	1	EMUDI	0	Emulator Data Dirty Interrupt flag 0 = DCEMU has not been written to since the clearing of this bit. 1 = DCEMU contains a new data value. This bit is cleared by writing this bit to 0.

Table 9-9. Debug Communication Status Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	1	CPURI	0	CPU Data Read Interrupt flag 0 = DCCPU has not been read since the clearing of this bit. 1 = DCEMU has been read. This bit is cleared by writing this bit to 0.
R/W	1	EMUD	0	Emulator Data Dirty 0 = DCEMU has not been written to since last read from CPU. 1 = DCEMU contains a new data value. This bit is cleared by reading DCEMU.
R/W	0	CPUD	0	CPU Data Dirty 0 = DCCPU has not been written to since last read from emulator. 1 = DCCPU contains a new data value. This bit is cleared by reading DCCPU.

9.2.14.6 Debug Communication Control Register (DCCR)

To enable the DCCPU read and DCEMU dirty interrupts the corresponding enable bits must be set in this register.

Table 9-10. Debug Communication Control Register

R/W	Bit Number	Field Name	Init. Val.	Description
R	31:2	Reserved	0x0000_ 0000	Reserved These bits are reserved, and will always read as 0
R/W	1	DCCPUIMASK	0	DCCPU Interrupt Mask 0 = DCCPU interrupts are disabled. 1 = DCCPU interrupts are enabled.
R/W	0	DCEMUIMASK	0	DCEMU Interrupt Mask 0 = DCEMU interrupts are disabled. 1 = DCEMU interrupts are enabled.

9.2.14.7 Development Control Register (DC)

DC is used for basic development control of the CPU.

Table 9-11. Development Control Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	31	ABORT	0	ABORT Writing ABORT to one while DBE is asserted causes the CPU to enter Debug Mode, regardless of SR:DM and any pending exceptions. If the CPU was in sleep mode, it will first be woken up before entering Debug Mode. The ABORT bit is cleared automatically when Debug Mode is entered.
S	30	RES	0	RES - Application Reset Writing this bit causes an application reset, which will reset the CPU and other system modules. The OCD state machines will be reset and the Transmit Queue flushed, but the OCD control and configuration registers will not be cleared.
R/W	29	MM	0	MM - Monitor Mode 1 = The CPU will enter Debug Mode in Monitor Mode 0 = The CPU will enter Debug Mode in OCD Mode Changing this bit in Debug Mode does not take effect until the CPU enters Debug Mode the next time.
R/W	28	ORP	0	ORP - OCD Register Protect 0 = OCD registers can be written by any privileged CPU mode 1= OCD registers can be written only in Debug Mode
R/W	27	RID	0	RID - Run In Debug 0: Peripherals are frozen in Debug Mode 1: Peripherals keep running in Debug Mode. In addition the PDBG register must be configured with individual masks for each module.
R	26	Reserved	0	
R/W	25	TOZ	0	TOZ - Trap Opcode Zero 0: The opcode 0x0000 is executed as a normal CPU instruction 1: The opcode 0x0000 causes entry to Debug Mode
R/W	24	IFM	0	IFM - Ignore First Match When written to one, a PC breakpoint on the first instruction after exiting Debug Mode with the *retd* instruction will not trigger re-entry to Debug Mode. Typically used when returning from a program breakpoint. This bit stays one until written to zero.

Table 9-11. Development Control Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	23	IRP	0	IRP - Instruction Replace If IRP is written to one before exiting OCD Mode with the *retd* instruction, the first instruction after exiting OCD Mode will be fetched from the Debug Instruction Register. This bit is cleared automatically after this fetch takes place. This bit will not have any effect if written at the same time as RES.
R/W	22	SQA	0	SQA - Software Quality Assurance 0: Regular program trace 1: SQA enhanced program trace
R/W	21:20	EOS	0	EOS - Event Out Select 00 = No operation 01 = Emit event out when the CPU enters Debug Mode 10 = Emit event out for breakpoints/watchpoints 11 = Emit event out for message insertion into the TXQ
R	19:14	Reserved		
R/W	13	DBE	0	DBE - Debug Enable DBE enables Debug Mode and all debug features in the CPU. DBE must be written to one to enable breakpoints, debug requests, or single steps.
R/W	12	DBR	0	DBR - Debug Request Writing DBR to one while DBE is asserted causes the CPU to enter Debug Mode. If the CPU was in sleep mode, it will first be woken up before entering Debug Mode. The DBR bit is cleared automatically when Debug Mode is entered.
	11:9	Reserved		
R/W	8	SS	0	SS - Single Step If SS is written to one before exiting Debug Mode with the retd instruction, exactly one instruction will be executed before returning to Debug Mode. SS stays one until written to zero by the debugger.

Table 9-11. Development Control Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	7:5	OVC	0	OVC[2:0] - Overrun Control OVC controls the action taken if Branch, Data, or Ownership trace messages are generated while the Transmit Queue is full. Settings 111 though 100 are reserved. 000 = Generate overrun messages 001 = Delay CPU to avoid BTM and Ownership Trace overruns 010 = Delay CPU to avoid DTM and Ownership Trace overruns 011 = Delay CPU to avoid BTM, DTM, and Ownership Trace overruns 111-100 = Reserved
R/W	4:3	EIC	0	EIC[1:0] - $\overline{\text{EVTI}}$ Control The EIC bits control the action performed when the $\overline{\text{EVTI}}$ pin on the Nexus debug port receives a high-to-low transition. If trace is enabled, $\overline{\text{EVTI}}$ can be configured to cause a trace synchronization message. If Debug Mode is enabled, $\overline{\text{EVTI}}$ can be configured to cause a breakpoint. 00 = $\overline{\text{EVTI}}$ for program and data trace synchronization 01 = $\overline{\text{EVTI}}$ for breakpoint generation 10 = No operation 11 = Reserved
R/W	2:0	TM	0	TM[2:0] - Trace Mode The TM bits select which trace modes are enabled. 000 = No Trace XX1 = OTM Enabled X1X = DTM Enabled 1XX = BTM Enabled If Data or Branch tracing is triggered or stopped by a watchpoint , the DTM and BTM bits are updated accordingly.

9.2.14.8 Development Status (DS) register

This register is used to examine the debug state of the CPU and the cause for entering Debug Mode. Note that multiple sources may trigger Debug Mode simultaneously, causing more than one bit to be set. The register is read-only. All bits are dynamic and do not require clearing.

This register is undefined when the CPU is not in Debug Mode.

Table 9-12. Development Status register

R/W	Bit Number	Field Name	Init. Val.	Description
R	31:29	Reserved	0	
R	28	NTBF	0	NTBF -NanoTrace Buffer Full This bit is set if Debug Mode is entered because the Memory Service Unit has signalled that the NanoTrace Buffer is full. This bit is cleared when Debug Mode is exited.
R	27	EXB	0	EXB -External Breakpoint This bit is set if Debug Mode was entered due to an event on the $\overline{\text{EVTI}}$ pin. This bit is cleared when Debug Mode is exited.
R	26	DBA	0	DBA - Debug Acknowledge This bit is set if Debug Mode was entered due to setting the Debug Request or ABORT bit in the DC register. This bit is cleared when Debug Mode is exited.
R	25	BOZ	0	BOZ - Break on Opcode Zero This bit is set if Debug Mode was entered due to opcode 0x0000 being executed. This bit is cleared when Debug Mode is exited.
R	24	INC	0	INC - Instruction Complete 0: The CPU is executing one or more instructions, or is not in OCD Mode. 1: The CPU is in OCD Mode and is not executing any instructions.
R	23:16	Reserved	0	
R	15:8	BP[7:0]	0	BP - Breakpoint Status The BP bits identify which hardware breakpoint caused Debug Mode to be entered: BP[0]: BP0A BP[1]: BP0B BP[2]: BP1A BP[3]: BP1B BP[4]: BP2A BP[5]: BP2B BP[6]: BP3A BP[7]: BP3B These bits are cleared when Debug Mode is exited.
R	7:6	Reserved	0	
R	5	DBS	0	DBS - Debug Status DBS is set when the CPU is in OCD Mode, otherwise cleared. This bit stays cleared also when the CPU operates in Monitor Mode.

Table 9-12. Development Status register

R/W	Bit Number	Field Name	Init. Val.	Description
R	4	STP	0	STP - Stop Status STP is set if OCD Mode is entered from sleep mode. This bit can be used by the debugger to determine the proper return sequence from OCD Mode. This bit is cleared when OCD Mode is exited.
R	3	Reserved	0	
R	2	HWB	0	HWB - Hardware Breakpoint Status This bit is set if Debug Mode was entered due to a hardware breakpoint. The BP[7:0] bits should be examined to determine the breakpoint(s) which triggered. This bit is cleared when Debug Mode is exited.
R	1	SWB	0	SWB - Software Breakpoint Status This bit is set if Debug Mode was entered due to a breakpoint instruction being executed. Returning from a software breakpoint may require special handling by the debugger. This bit is cleared when Debug Mode is exited.
R	0	SSS	0	SSS - Single Step Status This bit is set when Debug Mode is entered due to a single step. This bit is cleared when Debug Mode is exited.

9.2.14.9 Debug Instruction Register (DINST)

The Debug Instruction Register contains the instruction to be executed in OCD Mode. The CPU fetches and executes the instruction faster than they can be written by the Debug port. DINST is also used to store the instruction to replace the breakpoint instruction.

Table 9-13. Debug Instruction register

R/W	**Bit Number**	**Field Name**	**Init. Val.**	**Description**
R/W	31:0	DINST	0	DINST - Debug Instruction The instruction to be executed on the CPU.

9.2.14.10 Peripheral Debug Register (PDBG)

The Peripheral Debug Register controls the operation of modules in debug mode. If the DC.RID bit is set, the CPU is in debug mode and the PDBG bit for a module is set this module is kept running in debug mode. Otherwise the module is stopped. The mapping between the bits in this register and modules are part specific and are described in the OCD module configuration section of the part datasheet.

Table 9-14. Debug Instruction register

R/W	**Bit Number**	**Field Name**	**Init. Val.**	**Description**
R/W	31:0	PDBG	0	PDBG - Peripheral debug. 0 = The peripheral is running in debug mode. 1 = The peripheral is stopped in debug mode.

9.2.14.11 Debug Program Counter (DPC)

This register contains the PC value of the last executed instruction in any non-debug mode. This allows a debugger to sample program execution addresses for statistical purposes without interrupting the CPU.

If this register is read in Debug Mode, it will reflect the last executed instruction before Debug Mode was entered. Note that several types of breakpoints trigger before an instruction is executed, so this value is not necessarily identical to RAR_DBG.

When replacing the return instruction from Debug Mode, the CPU will see the DPC value as the PC value for the executed instruction. The user only needs to write this register when replacing the return instruction from OCD Mode.

Table 9-15. Debug Program Counter

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	31:0	DPC	0	DPC - Debug Program Counter PC of the last executed instruction

9.3 Debug Port

9.3.1 Overview

The OCD debug port consists of the JTAG port and the AUX port. The low bandwidth JTAG port handles all register access, while the high bandwidth AUX port transfers all Nexus messages from the OCD system.

The Nexus standard defines the maximum clock frequency for JTAG to be 33 MHz, and for AUX 200 MHz.

9.3.2 JTAG

Access to OCD register is done through an IEEE1149.1 JTAG-port. The JTAG TAP controller is shared with the rest of the system. In order to enable access to OCD register the emulator must perform the following sequence.

1. Put the TAP controller in the state "test logic reset".
2. Insert the OCD Instruction to prepare the Debug Port to receive OCD register access. The OCD instruction is inserted using the IR scan path.
3. Use the DR scan path to insert the OCD register address and operation (Read / Write).
4. Use the DR scan path to read / write the data to / from the register.
5. Repeat 3 through 4 for every register operation. The TAP controller will remain in OCD mode until a test logic reset is detected.

To be able to use JTAG-based debug tools for AVR32 without adapters, it is recommended that a circuit design using an AVR32 device should use a standard 10-pin 50-mil IDC connector with the pinout shown in Table 9-16. The signals are described in Table 9-17.

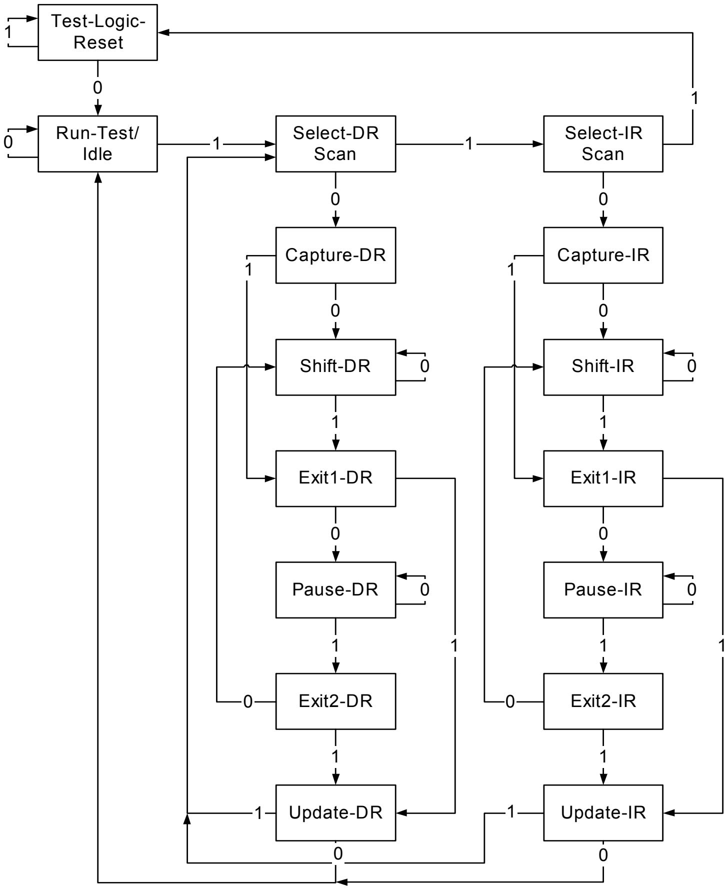
Table 9-16. AVR32 standard JTAG connector pinout. All directions relative to processor

Signal	Dir	Pin	Pin	Dir	Signal
TCK	In	1	2		GND
TDO	Out	3	4	Out	VREF
TMS	In	5	6	In	RESET_N
N/C		7	8		N/C
TDI	In	9	10		N/C

Table 9-17. JTAG signals

Pin	Direction	Description
TRST_N	Input	Asynchronous reset for the TAP controller and JTAG registers
TCK	Input	Test Clock. Data is driven on falling edge, sampled on rising edge.
TMS	Input	Test Mode Select
TDI	Input	Test Data In
TDO	Output	Test Data Out
RESET_N	Input	Device reset
VREF	Output	Reference voltage from target. Signals should be driven relative to this voltage level.

Figure 9-4. JTAG TAP controller state diagram.



9.3.3 AUX port

The Auxiliary (AUX) port and messaging protocol follow the definitions of the Nexus standard. This standard allows varying the number of signalling pins. The following configuration is selected for AVR32UC.

- 6 data output pins (MDO)
- 2 message start/end output pins ($\overline{\text{MSEO}}$)
- 1 $\overline{\text{EVTO}}$ pin
- 1 $\overline{\text{EVTI}}$ pin

The configuration is based on the presumed needs for bandwidth in a system being traced at 100+ MIPS, balanced against the desire to keep debug pincount low. This configuration can be changed in future implementations to allow for greater or smaller bandwidth over the AUX port.

The AUX pins may be multiplexed with GPIO in a device. By default, the MCKO, MDO, and $\overline{\text{MSEO}}$ pins are tristated or used as GPIO, and the Nexus functionality must be explicitly enabled by the debugger. $\overline{\text{EVTO}}$, $\overline{\text{EVTI}}$, and the JTAG pins are always available to the debugger.

If the AUX pins are needed for Nexus functionality in an application, it is recommended not to use these pins for GPIO purposes, as this can affect the signal integrity required for Nexus operation.

The complete signal list of the AUX port is shown in Table 9-18.

Table 9-18. Auxiliary pins

Auxiliary pins	**Width**	**Direction**	**Description**
MCKO	1	O	Message Clockout (MCKO) is a free-running output clock to development tools for timing of MDO and $\overline{\text{MSEO}}$ pin functions.
MDO	6	O	Message Data Out (MDO[5:0]) are output pins used for all messages generated by the device. In single datarate mode, external latching of MDO shall occur on rising edge of MCKO. In double datarate mode, external latching of MDO shall occur on both edges of MCKO.
MSEO	2	O	Message Start/End Out ($\overline{\text{MSEO[1:0]}}$) pins indicate when a message on the MDO pins has started, when a variable length packet has ended, and when the message has ended. In single datarate mode, external latching of $\overline{\text{MSEO}}$ shall occur on rising edge of MCKO. In double datarate mode, external latching of $\overline{\text{MSEO}}$ shall occur on both edges of MCKO.
EVTO	1	O	Event Out ($\overline{\text{EVTO}}$) is an output pin which can be configured to toggle every time a message is inserted into the Transmit Queue, when the CPU entered OCD Mode, or when a breakpoint or watchpoint hit occured, as configured by the EOS bits in the Development Control register .
EVTI	1	I	Event In ($\overline{\text{EVTI}}$) is an input which, when a high-to-low transition occurs, a processor is halted (breakpoint) or program and data synchronization messages are transmitted from the OCD controller, as configured by the EIC bits in the Development Control register.
RESET_N	1	I	System reset

To be able to use AUX-based debug tools for AVR32, a circuit design using an AVR32 device should use a Mictor38 connector (AMP P/N 767054-1) as defined in the Nexus standard, with the pinout shown in Table 9-19.

Table 9-19. AVR32 standard Nexus connector pinout. All directions relative to processor

Signal	Dir	Pin	Pin	Dir	Signal
MSEO0	Out	38	37		N/C
MSEO1	Out	36	35		N/C
MCKO	Out	34	33		N/C
EVTO_N	Out	32	31		N/C
MDO0	Out	30	29		N/C
MDO1	Out	28	27		N/C
MDO2	Out	26	25		N/C
MDO3	Out	24	23		N/C
MDO4	Out	22	21	In	TRST_N
MDO5	Out	20	19	In	TDI
	N/C	18	17	In	TMS
	N/C	16	15	In	TCK
	N/C	14	13		N/C
VREF	Out	12	11	Out	TDO
EVTI_N	In	10	9	In	RESET_N
	N/C	8	7		N/C
	N/C	6	5		N/C
	N/C	4	3		N/C
	N/C	2	1		N/C

9.3.3.1 *Reset configuration*

The Nexus standard specifies that the AUX port can be enabled by keeping $\overline{\text{EVTI}}$ low while pulsing $\overline{\text{TRST}}$ (or exiting Test-Logic-Reset). The OCD system in AVR32 has removed this feature. In order to enable the AUX port, the debugger has to write the AXC:AXE (Auxiliary Enable) bit.

9.3.3.2 *Message protocol*

The OCD System implements the Auxiliary Port Message Protocol defined in the Nexus standard. The following section is merely a summary of this protocol. For details, please see the Nexus standard.

Messages are composed of a Start-of-Message (SOM) token, followed by one or more packets of information, each of fixed or variable length, and ended by an End-of-Message (EOM) token. SOM/EOM and End-of-Variable-Length-Packets (EVLP) are signalled by $\overline{\text{MSEO}}$ for transmitted messages. Packet information is carried by the MDO pins. The number of MDO pins available is known as the *port boundary*. The information carried by the MDO and MSEO pins each cycle is known as a *frame*.

9.3.3.3 *Message rules*

MDO is valid whenever MSEO does not indicate "idle".

Fixed length packets are implicitly recognized from the message format, and are not required to end on a port boundary. Thus, packets may also start within a port boundary if following a fixed length packet. The end of variable length packets is identified through the $\overline{\text{MSEO}}$ pins, and to identify the end of the packet uniquely, these packets must end on a port boundary. If necessary, the packet must be stuffed with zeroes to align the end to a port boundary. Variable length packets may be truncated by omitting leading zeroes so that the packet ends on the first possible port boundary.

- The $\overline{\text{MSEO}}$ pins behave the following way ("x" means "don’t care"):
- 0b11 followed by 0b00 indicates SOM
- 0b0x followed by 0b11 indicates EOM
- 0b00 followed by 0b01 indicates EVLP
- $\overline{\text{MSEO}}$ is 0b00 at all other clocks during transmission of a message
- $\overline{\text{MSEO}}$ is 0b11 at all clocks when idle.

9.3.3.4 *Clock and frame rate*

In single datarate mode (default), MDO and MSEO should be sampled by an external tool on the rising edge of MCKO. In double datarate mode, the MCKO clock runs at half frequency, so MDO and MSEO should be sampled on both edges of MCKO. This is configured by the Double Datarate bit in the AUX Port Control Register.

It is also possible to reduce the frequency of the AUX port compared to the CPU clock by writing the AXC:LS and AXC:DIV bits. If LS=1, the DIV value selects the frame rate of the AUX port:

$f_{AUX} = f_{CPU}/(DIV+1)$

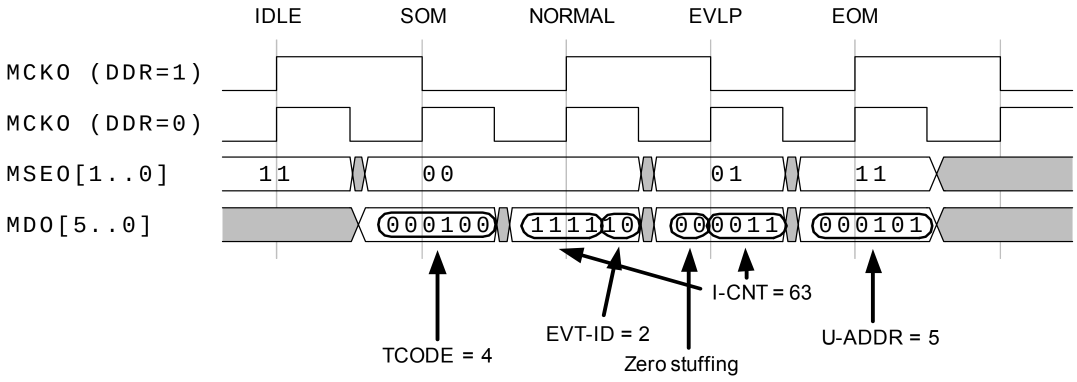
If LS=1 and DIV=0, $f_{AUX} = f_{CPU}/2$.

This can be combined with the single or dual datarate mode, as described above. In either case, the sampling edge will be as close to the middle of the MDO data frame as possible. The duty cycle of the MCKO clock will stay within the 40-60 duty cycle requirement of the Nexus standard for all settings apart from DIV=2.

9.3.3.5 *Example*

Figure 9-5 shows an example of transmission of a Program Trace Indirect Branch message. The TCODE is fixed at 6 bits (=4 for PTIB), followed by a fixed-length packet (EVT-ID = 2), and a variable-length packet (I-CNT = 63). I-CNT is stuffed with zeroes to fit the port boundary. Finally, the variable packet U-ADDR (=5) is transmitted. Since this leading zeroes of this packet can be truncated, it fits within a single frame.

Figure 9-5. Example of a Nexus message transmission with single and double datarate.



9.3.3.6 *Transmit queue and overruns*

Messages from various sources are inserted in a Transmit Queue (TXQ), which stores a number of frames. This queue acts as a FIFO which allows messages to be inserted more rapidly than they can be retrieved by the emulator.

The queue holds 16 frames. If more messages are inserted than there is room for in the queue, information will be lost, and an overrun situation occurs. The TXQ will block any more messages from being inserted, and allow the queue to be emptied by the emulator before allowing any more messages to be inserted. The first message to be inserted after the overrun is cleared, is an Error message, which informs the emulator that an overrun has occurred and which types of trace messages have been lost. After this, transmission continues as normal.

Alternatively, the user can configure the OCD to halt the CPU to prevent overruns. This can be done selectively for different message types, and is controlled by writing to the Overrun Control (OVC) bits in the DC register.

If any of the OVC bits are set, watchpoint trace messages will usually not generate TXQ overflow. However, triggering an program and data watchpoint on the same instruction may in some rare cases cause an overrun independently of the OVC settings, since a large amount of trace message data will be produced for this instruction.

9.3.3.7 *Trace and reset*

All pending trace messages in the Transmit Queue are flushed if: the OCD is reset by a system reset; the OCD is disabled; or an application reset is triggered by writing to the DC:RES bit.

Thus, if the CPU is reset, but not the OCD, the program flow can be observed by program trace. However, if the debugger resets the system, the remaining messages in the queue are of no value, and expected to be flushed.

Note that if the OCD is disabled (by clearing DC:DBE or by a system reset), trace is suspended until DC:DBE is written to one. The DC:TM bits must be written simultaneously, and define which trace features should now be active.

Similarly, when an application reset is triggered by writing DC:RES, the DC:TM bits are written simultaneously and define which trace features should now be active.

9.3.4 Messages

9.3.4.1 Error

The error message indicates various errors that can occur during trace or debugging. Table 9-21 lists the various errors that can be reported, along with the associated ECODE.

If trace messages are lost because of insufficient space in the Transmit Queue, an error message is transmitted, followed by a synchronization message, as soon as space is available in the Transmit Queue.

Table 9-20. Error

Indirect Branch Message with Sync			**Direction: From target**
Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
5	ECODE	Fixed	Error code. Refer to Table 9-21.
6	TCODE	Fixed	Value = 8

Table 9-21. Error codes

ECODE	**Description**
0b00000	Ownership trace overrun
0b00001	Program trace overrun
0b00010	Data trace overrun
0b00011 - 0b00101	Reserved
0b00110	Watchpoint overrun.
0b00111	Program and/or data and/or ownership trace overrun.
0b01000	Program trace and/or data and/or ownership trace and/or watchpoint overrun.
0b01001 - 0b11111	Reserved

9.3.5 Registers

9.3.5.1 Auxiliary Port Control Register (AXC)

Table 9-22 shows the description of the Auxiliary Port Control Register. This register allows greater flexibility in controlling the operation of the AUX port than specified by the Nexus standard. This includes enabling the AUX port, and controlling the speed of the clock and data compared to the CPU clock.

Table 9-22. AUX Port Control Register

R/W	Bit Number	Field Name	Init. Val.	Description
R	31:16	Reserved	0	Reserved These bits are reserved, and will always read as 0
R/W	15:14	AXS	0	AXS - Auxiliary Port Select 0: AUX port is mapped to pin configuration 0. 1: AUX port is mapped to pin configuration 1. 2: AUX port is mapped to pin configuration 2. 3: AUX port is mapped to pin configuration 3.
R	13	Reserved	0	Reserved This bit is reserved, and will always read as 0
R	12	Reserved	0	Reserved This bit is reserved, and will always read as 0.
R/W	11	LS	0	LS - Low Speed 0:AUX port runs at the same speed as the CPU 1:AUX port runs at reduced speed compared to the CPU.
R/W	10	DDR	0	DDR - Double Data Rate Setting this bit halves the MCKO rate so that MDO data must be sampled on both edges of MCKO. 1 = Double data rate mode 0 = Single datarate mode
R/W	9	AXO	0	AXO - Auxiliary Port Override 0: AUX port is mapped to the pins dictated by AXS. 1: AUX port is overridden and mapped to pin configuration 1.
R/W	8	AXE	0	AXE - Auxiliary Port Enable 0: AUX port is used for GPIO 1: AUX port is used for Nexus operation. This bit does not need to be written in devices with dedicated AUX pins
R	7:4	Reserved	0	Reserved These bits are reserved, and will always read as 0
R/W	3:0	DIV	0	DIV - Division factor If LS=1, the DIV value selects the frame rate of the AUX port.

9.4 Breakpoints

9.4.1 Overview

The Nexus Recommended Register map supports up to 8 universal breakpoints. However since the AVR32UC hardware employs separate instruction and data memories, the OCD system must also separate program and data breakpoints. Any breakpoint can also be programmed as a watchpoint. The watchpoint will trigger a Watchpoint Hit message. The OCD system supports up to six program breakpoints modules and two data breakpoint modules. In addition to this, the data trace modules can also be used as data address watchpoints. The trace watchpoints result in a vendor defined Trace Watchpoint Hit message.

Figure 9-6. Breakpoint modules.

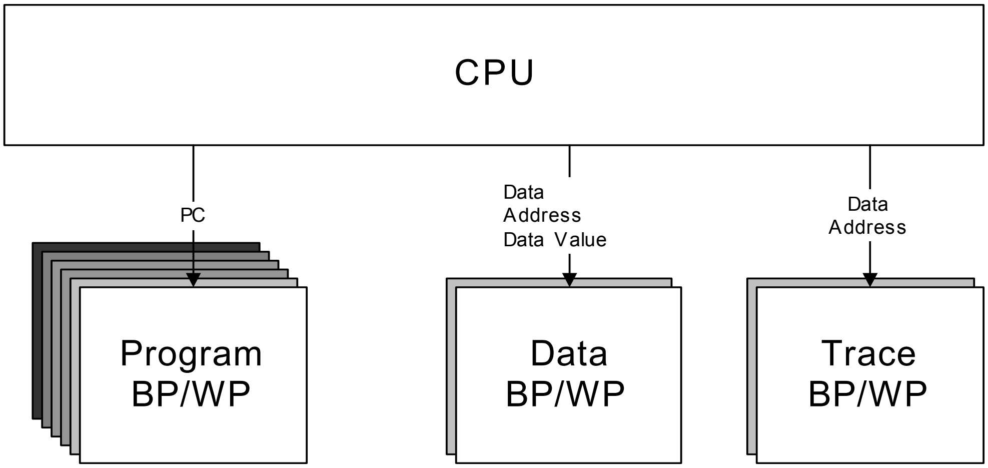
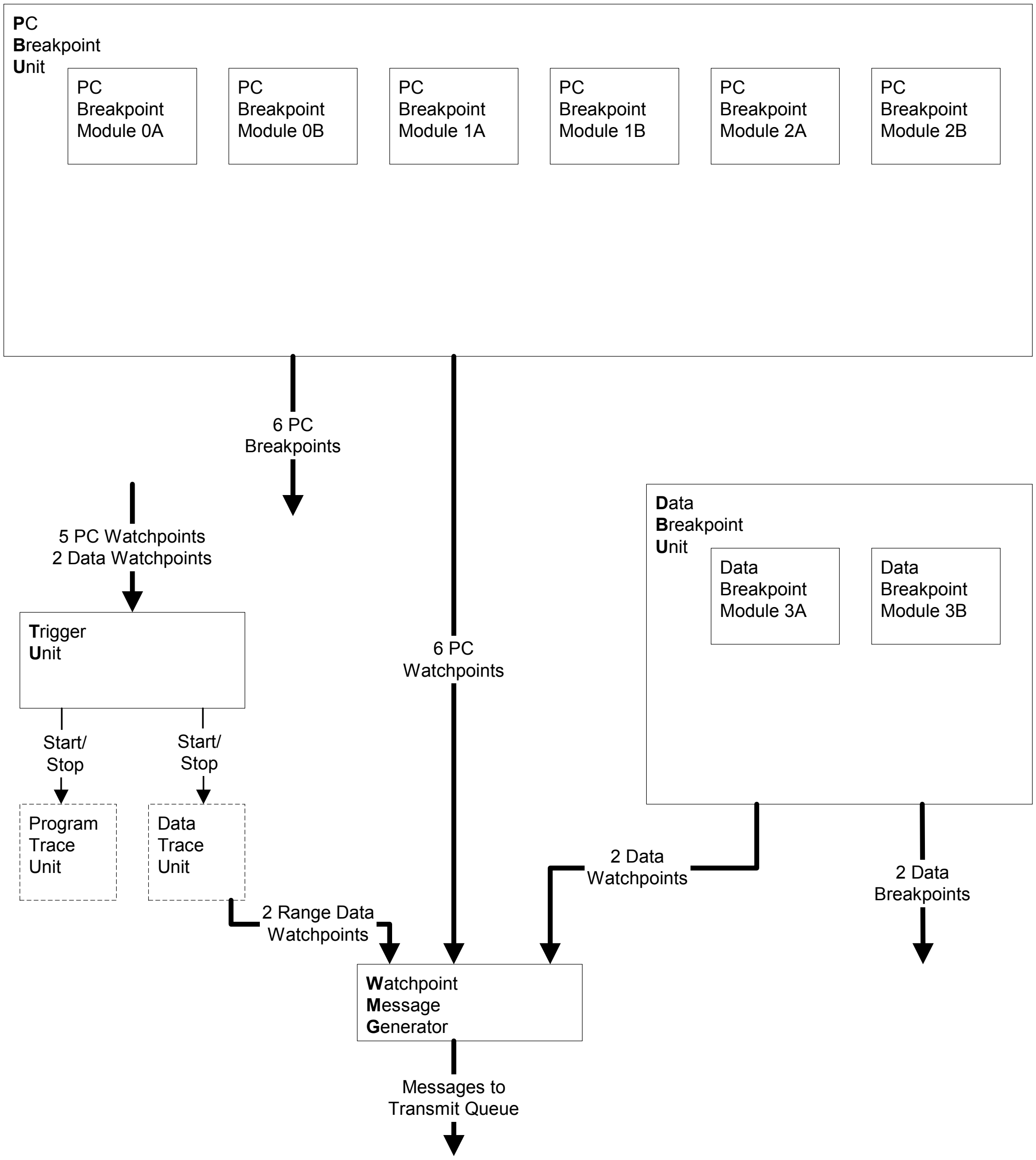


Figure 9-7. Breakpoint unit overview.



9.4.2 Breakpoint Unit description

The Breakpoint unit consists of the units shown in Figure 9-7. The PC Breakpoint Unit (PBU) handles the program counter breakpoints. The PBU can have up to 6 PC breakpoint modules that can match on a single PC. Two modules can be combined to give a match on a range of PC values, thus up to three ranges can be defined. The PBU is configured with registers Breakpoint / Watchpoint Control (BWC) and Breakpoint / Watchpoint Address (BWA) 0A, 0B, 1A, 1B, 2A, and 2B.

The Data Breakpoint Unit handles data breakpoints. The data breakpoints can be configured with the BWC / BWA / BWD 3A and 3B registers.

The Watchpoint Message Generator (WMG) generates watchpoint messages for all breakpoint modules and data trace watchpoints.

Optionally, a breakpoint or watchpoint can be signalled by a pulse on the $\overline{\text{EVTO}}$ pin. This requires DC:EOS bits to be set to 1 and EOC in the corresponding Breakpoint/Watchpoint Control Register must be written to one.

9.4.2.1 Program Breakpoints

In order to enable a simple program breakpoint the Breakpoint / Watchpoint Address (BWA) and Breakpoint / Watchpoint Control (BWC) registers for that breakpoint must be updated.

The BWA register must be written with the address of the instruction where the debugger wants to halt.

The BWC must have the Breakpoint / Watchpoint Enable (BWE) field set to breakpoint.

Program breakpoints break on the instruction pointed to by BWA. The instruction will cause a debug exception and the Debug Mode Link Register (RAR_DBG) and Debug Mode Return Status Register (RSR_DBG) will point to the instruction that caused the debug exception. The Development Status register will also be updated to indicate which breakpoint caused the exception. In OCD Mode the debug tool can then feed the CPU with debug code to ascertain the state of the processor. In OCD Mode the breakpoint modules are disabled.

Upon return from Debug Mode, the PC and SR will be restored from the RAR_DBG and RSR_DBG and the instruction that caused the debug exception will be fetched again. If the program breakpoint has not been disabled in Debug Mode, the Ignore First Match (IFM) bit in the Development Control (DC) register must be written to one to avoid triggering another breakpoint on the first instruction after exiting Debug Mode. The IFM bit prevents any Program Breakpoint operation on the first instruction after exiting Debug Mode.

9.4.2.2 Watchpoints

When enabled in the BWC, a watchpoint message is sent when the instruction address matches the address stored in BWA. If both a Trace watchpoint and a Watchpoint triggers at the same time, the Trace watchpoint will be ignored and only a Watchpoint Hit message will be generated.

Note that Program, Data, and Trace watchpoints are generated at different pipeline stages and will not be synchronized when the messages are generated. A Program Watchpoint on a load store instruction will hit before a data watchpoint on the same instruction.

9.4.2.3 Data Breakpoints

Data Breakpoint modules listen on the data address and data value lines between the CPU and the data cache and can halt the CPU, or send a watchpoint message, if the address and / or value meets a stored compare value. Unlike program breakpoints, data breakpoints halt on the next instruction after the load / store instruction that caused the breakpoint has completed.

The BWA register must be written with the address of the data the debugger wants to halt on.

9.4.3 Data Breakpoint interface

9.4.3.1 Data alignment

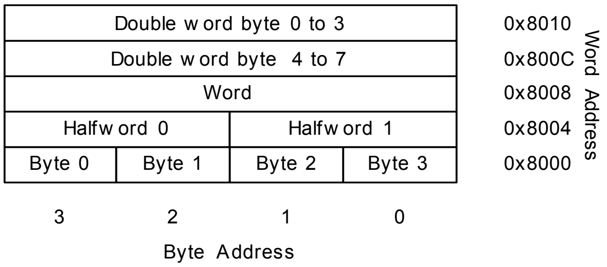
The AVR32 can read or write data in bytes, halfwords, or words. The same data location can be accessed through either operation, e.g. a byte location can be accessed as part of a double word. The data bus operations seen by the OCD system are always aligned, i.e. halfwords start on halfword boundaries, word accesses start on word boundaries, as illustrated in Figure 9-8. If the data bus operation is a double word load / store, the breakpoint module will see the word data value which corresponds to the address in BWA.

One data breakpoint module can only compare 32 bits of data. The data to be matched can therefore not cross a word boundary if the data breakpoint is to match correctly. When the debugger wants to match on a byte or halfword, the BWD register must be written with the LSB aligned, and the BWC:BME bits must be set to mask the upper bits of the BWD register.

For example, if the debugger wants to match against Byte 1 in Figure 9-8, the BWA must be set to the byte address of Byte 1 and the BWD written with the value to match on aligned to LSB. Also the BWC:BME must be set to mask the 24 most significant bits of the BWD register (BME = 0xE).

By default, the data breakpoint module will match on the data value regardless of the size of the access. The data BWC can also be set to match on a specific access size if the SIZE bits are set. The debugger can for example, set the breakpoint module to match only on byte writes to byte 1 in Figure 9-8. The BWD register must still be aligned correctly, and the byte mask must be set, but the data breakpoint will only trigger if a single byte is written to byte 1 and not if, for example, a whole word is written to byte 0, 1, 2, and 3.

Figure 9-8. Memory access data alignment.



9.4.4 Triggering trace

A watchpoint from the program or data breakpoint modules can be used to start or stop program or data trace. This is done using a trigger unit. The trigger unit can be configured using the watchpoint trigger register. When the trigger unit is set to start trace upon a watchpoint, DC:TM will be set accordingly, and trace will then be enabled. If a data watchpoint enables data trace, the data event is not included in the data trace output, while an event which disables data trace is included in the data trace output.

9.4.5 Messages

9.4.5.1 Watchpoint Hit (WH)

Table 9-23. Watchpoint Hit

Watchpoint Message			**Direction: From target**
Packet Size	**Packet Name**	**Packet Type**	**Description**

8	WPHIT	Fixed	XXXXXXX1 = Watchpoint 0 matched
			XXXXXX1X = Watchpoint 1 matched
			...
			X1XXXXXX = Watchpoint 6 matched
			1XXXXXXX = Watchpoint 7 matched

6	TCODE	Fixed	Value = 15

9.4.5.2 Trace Watchpoint Hit (TWH)

Table 9-24. Trace Watchpoint Hit

Trace Watchpoint Message			**Direction: From target**
Packet Size	**Packet Name**	**Packet Type**	**Description**
2	WPHIT	Fixed	X1 = Watchpoint 0 matched 1X = Watchpoint 1 matched
6	TCODE	Fixed	Value = 56

9.4.6 Registers

9.4.6.1 PC Breakpoint/Watchpoint Address registers (BWA0A, BWA0, BWA1A, BWA1B, BWA2A, BWA2B)

The 6 BWA registers contains one instruction address each. The address can be used for a single breakpoint match or used as bitwise mask to create a range.

Table 9-25. PC BWAnx Register

R/W	**Bit Number**	**Field Name**	**Init. Val.**	**Description**
R/W	31:0	BWA	0	Breakpoint/Watchpoint Address

9.4.6.2 PC Breakpoint/Watchpoint Control registers - (BWC0A, BWC0B, BWC1A, BWC1B, BWC2A, BWC2B)

Table 9-26. PC BWCnx Register

R/W	Bit Number	Field Name	Init. Val.	Description
RW	31:30	BWE	00	BWE - Breakpoint / Watchpoint Enable 00 = Disabled 01 = Breakpoint enabled 10 = Reserved 11 = Watchpoint enabled
R	29:26	Reserved	0	Reserved
RW	25	AME	0	AME - Address Mask Enable This bit is only present in BWCxA registers. 0 = Disabled. 1 = Enabled. BWAxB will be used to bitwise mask the PC compare according to this function: BP A: (PC & BWA_B) == (BWA_A & BWA_B) BP B: Will never trigger
R	24:15	Reserved	0	Reserved
RW	14	EOC	0	EOC - EVTO Control 0 = Breakpoint/watchpoint status indication is not output on EVTO 1 = Breakpoint/watchpoint status indication is output on EVTO
R	13:0	Reserved	0	Reserved

9.4.6.3 Data Breakpoint / Watchpoint Address (BWA3A, BWA3B)

Table 9-27. Data Breakpoint/Watchpoint address (BWA3x) register

R/W	Bit Number	Field Name	Init. Val.	Description
RW	31:0	BWA	0x00000000	Address of data for breakpoint or watchpoint generation.

9.4.6.4 Data Breakpoint / Watchpoint Data (BWD3A, BWD3B)

Table 9-28. Data Breakpoint/Watchpoint data (BWD3x) register

R/W	Bit Number	Field Name	Init. Val.	Description
RW	31:0	BWD	0x00000000	Data value for breakpoint or watchpoint generation.

9.4.6.5 Data Breakpoint / Watchpoint Control (BWC3A, BWC3B)

Table 9-29. Data Breakpoint / Watchpoint Control (BWC3x)

R/W	Bit Number	Field Name	Init. Val.	Description
RW	31:30	BWE	00	BWE - Breakpoint / Watchpoint Enable 00 = Disabled 01 = Breakpoint enabled 10 = Reserved 11 = Watchpoint enabled
RW	29:28	BRW	00	BRW - Breakpoint/Watchpoint Read/Write Select 00 = Break on read access 01 = Break on write access 10 = Break on any access 11 = Reserved
R	27:24	Reserved	00	Reserved

RW	23:20	BME	0x0	BME - Breakpoint/Watchpoint Data Mask
				1XXX = Mask bits 31:24 in BWD
				X1XX = Mask bits 23:16 in BWD
				XX1X = Mask bits 15:8 in BWD
				XXX1 = Mask bits 7:0 in BWD

R	19:18	Reserved	00	Reserved
RW	17:16	BWO	000	BWO - Breakpoint/Watchpoint Operand 1X = Compare with BWA value X1 = Compare with BWD value
R	15	Reserved	0	Reserved
RW	14	EOC	0	EOC - EVTO Control 0 = Breakpoint/watchpoint status indication not output on EVTO 1 = Breakpoint/watchpoint status indication is output on EVTO
R	13:12	Reserved	0	Reserved
R/W	11:9	SIZE	000	SIZE - Size bits to match 0xx = Disregard access size (Default) 100 = Byte access 101 = Halfword access 110 = Word access 111 = Reserved
R/W	8:0	Reserved	0	Reserved

9.4.6.6 Watchpoint Trigger

Table 9-30. WT, Watchpoint Trigger Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	31:29	PTS	000	PTS - Program Trace Start 000 = Trigger disabled 001 = Program watchpoint 0b 010 = Program watchpoint 1a 011 = Program watchpoint 1b 100 = Program watchpoint 2a 101 = Program watchpoint 2b 110 = Data watchpoint 3a 111 = Data watchpoint 3b
R/W	28:26	PTE	000	PTE - Program Trace End 000 = Trigger disabled 001 <-> 111 Watchpoint selected as for PTS
R/W	25:23	DTS	000	DTS - Data Trace Start 000 = Trigger disabled 001 <-> 111 Watchpoint selected as for PTS
R/W	22:20	DTE	000	DTE - Data Trace End 000 = Trigger disabled 001 <-> 111 Watchpoint selected as for PTS
R	19:0	Reserved	-	Reserved

9.5 Program trace

9.5.1 Program trace overview

The AVR32 OCD system provides program trace support via the debug port. The program trace feature implements a Program Flow Change Model in which the program trace is synchronized at each program flow discontinuity. This occurs at taken indirect branches and exceptions. A record of taken / not taken direct branches is included so that the complete program flow can be decoded.

The development tool can then interpolate what transpires between each program trace message by correlating information from branch target messaging and static source or object code files. Self-modifying code cannot be traced with the Program Flow Change Model because the source code is not static.

The TM[2] bit in the Development Control register must be set to enable program trace.

9.5.1.1 Branch message summary

Five types of branch messages can be generated:

1. Program Trace, Indirect Branch is transmitted on most subroutine calls, returns, interrupts, exceptions, and any situation where the target address of a branch cannot be determined from the source code. This message contains the instruction count to identify the branch and the target PC to identify the branch target.
2. Program Trace Synchronization is transmitted to indicate the current PC after starting trace or after trace synchronization is lost.

3. Program Trace, Indirect Branch messages with sync contain both instruction count and PC, and are transmitted instead of a Program Trace Synchronization message if a synchronization condition occurs and the current instruction is a taken direct/indirect branch.
4. Program Trace, Resource full messages is transmitted when an internal buffer overflows. ICNT is transmitted whenever it overflows with this message.
5. Program Trace Correlation. This message is transmitted to synchonize the program trace with an event. Sent when trace is disabled, debug mode is entered or sleep mode is entered.

The Nexus standard also specifies Program Trace Correction messages to correct for speculatively transmitted trace messages, but these are not implemented in the AVR32, since program trace messages are only transmitted for actually executed instructions. Similarly, the Nexus-specified CANCEL packet of synchronized branch messages is not implemented in AVR32.

Entry into Debug Mode will generate an program trace correlation message, while no trace messages are generated while executing in Debug Mode. A Program Trace Synchronization message is transmitted when Debug Mode is exited.

9.5.2 Branch message packets

The program trace messages contain packets which identify the address of the taken branch, the target of the branch, and the current program counter value. These packets are discussed below.

9.5.2.1 Instruction count packet

In several of the program trace messages, an Instruction Count (I-CNT) packet is included, to identify the number of sequentially executed instruction units since the last program trace message. In AVR32, this figure refers to bytes, i.e. compact instructions count two bytes and extended instructions are four bytes.

The following rules apply to instruction counts:

- A taken indirect branch which generates a trace message is not included in the instruction count.
- An indirect branch which is not taken is included in the instruction count.
- Speculatively fetched instructions are not counted until they are actually executed.
- The instruction counter is reset every time a program trace message is generated.

9.5.2.2 Compressed program counter packets

To save bandwidth, the Nexus messages employ compressed versions of the program counter address. These include:

U-ADDR = StripLeadingZeros (Previous sent addr xor uncompressed address from pipeline).

F-ADDR = Full target address for a taken branch. Leading zeroes may be truncated.

9.5.3 Special cases

9.5.3.1 Debug Mode

When entering Debug Mode, a PTC message is generated with EVCODE = 0.

When exiting Debug Mode, a PTSY message is generated. If the instruction also generates a branch message, the branch message with sync (i.e. PTDBS or PTIBS) is generated instead of

PTSY. In this case, the address of the instruction which generated the branch message can not be explicitly reconstructed from the trace log, but the debugger will normally know which address was returned to when Debug Mode was exited.

If a breakpoint occurs on the first instruction after exiting Debug Mode, a PTC message with EVCODE = 0 is generated.

9.5.4 Messages

9.5.4.1 Program Trace, Direct Branch

This message is output by the target processor whenever there is a change of program flow caused by a conditional or unconditional branch. The instruction count (I-CNT) is included to identify the branch address. The following AVR32 instructions can cause a direct branch:

Table 9-31. Direct branch instructions

Mnemonic		Description
br{cond3}	Compact	Branch if condition satisfied.
br{cond4}	Extended	
rjmp	Compact	Branch if condition satisfied.

Table 9-32. Direct Branch message without sync

Direct Branch Message			Direction: From target
Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	I-CNT	Variable	Number of bytes executed since the last taken branch.
6	TCODE	Fixed	Value = 3

9.5.4.2 Program Trace, Direct Branch with Target Address

This message is transmitted instead of the Direct Branch message when SQA enhanced program trace is enabled by writing DC:SQA to one. This simplifies real-time PC reconstruction in the emulator for real-time code coverage and performance analysis purposes.

Table 9-33. Direct Branch message with Target Address

Direct Branch Message with Sync			Direction: From target
Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
32	U-ADDR	Variable	The unique portion of the branch target address for a taken indirect branch or exception. Most significant bits that have a value of 0 are truncated.
8	I-CNT	Variable	Number of bytes executed since the last taken branch.
6	TCODE	Fixed	Value = 57

9.5.4.3 Program Trace, Indirect Branch

An indirect branch is output by the target processor whenever there is a change of program flow caused by a subroutine call, return instruction, interrupt, or exception.

Messages for taken indirect branches and exceptions include how many sequential bytes were executed since the last taken branch or exception, and the unique portion of the branch target address or exception vector address. The unique portion of the branch is found by doing an exclusively or on the branch target and the last sent UADDR / FADDR. Additionally, the cause of the indirect branch is identified through an Event ID packet. Operations causing indirect branches and their corresponding EVT-ID are shown below.

Table 9-34. Operations causing indirect branch messages

Description	Operation	EVT-ID
Exception entry	Exception, interrupts (0 to 3), NMI, entry to Debug Mode	3
Subroutine call	acall, icall, mcall, jcall, scall, rcall instruction	2
Branch via register contents	Any mov (except mov pc, lr) or load (except popm/ldm) with PC as destination. Any arithmetic instruction with PC as destination.	1
Return	ret{cond4}, rete, rets, retj, (mov pc, lr), popm/ldm loading PC	0

Note that subrotine returns are often accomplished by a *mov pc, lr*, *popm* or *ldm* instruction with PC included in the argument list. This generates an EVT-ID of 0 instead of 1..

Table 9-35. Indirect branch message without sync

Indirect Branch Message			**Direction: From target**
Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
32	U-ADDR	Variable	The unique portion of the branch target address for a taken indirect branch or exception. Most significant bits that have a value of 0 are truncated.
8	I-CNT	Variable	Number of bytes executed since the last taken branch.
2	EVT-ID	Fixed	Cause of indirect branch: 3: Exception entry 2: Call 1: Branch via register contents 0: Return
6	TCODE	Fixed	Value = 4

9.5.4.4 *Program Trace Synchronization*

This message is output by the PTU when any of the following conditions occurs:

1. Upon exit from reset. This is required to allow the number of instruction units executed packet in a subsequent Program Trace Message to be correctly interpreted by the tool.
2. When program trace is enabled during normal execution of the embedded processor.
3. Upon exit from a power-down state. This is required to allow the number of instruction units executed packet in a subsequent Program Trace Message to be correctly interpreted by the tool.
4. Upon exiting from Debug Mode.
5. An overrun condition had previously occurred in which one or more branch trace occurrences were discarded by the target processor's debug logic.To inform the tool that an overrun condition occurred, the target outputs an Error Message (TCODE = 8) with an

ECODE value of 00001 or 00111 immediately prior to the Program Trace Synchronization Message.

6. A debug control register field specifies that $\overline{\text{EVTI}}$ pin action is to generate program trace synchronization, and the Event-In ($\overline{\text{EVTI}}$) pin has been asserted.
7. Upon overflow of the sequential instruction unit counter.
8. After 256 branch messages without sync.

Table 9-36. Program Trace Synchronization Message

Program Trace Sync Message			Direction: From target
Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
32	PC	Variable	The full current instruction address. Most significant bits that have a value of 0 are truncated.
8	I-CNT	Variable	Number of bytes executed since the last taken branch.
6	TCODE	Fixed	Value = 9

9.5.4.5 Program Trace, Direct Branch with Sync

If a Program Trace Synchronization message occurs on an instruction which transmits a direct branch message, the Direct Branch with Sync message is transmitted instead of the Program Trace Synchronization message. The Direct Branch with Sync message contains the instruction count referring to the taken branch, as well as the complete PC value of the branch target.

The format for direct branch messages with sync is shown below. The AVR32 OCD system never issues speculative branch messages and there is therefore no CANCEL packet.

Table 9-37. Direct Branch message with Sync

Direct Branch Message with Sync			Direction: From target
Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
32	F-ADDR	Variable	The full target address for a taken direct branch. Most significant bits that have a value of 0 are truncated.
8	I-CNT	Variable	Number of bytes executed since the last taken branch.
6	TCODE	Fixed	Value = 11

9.5.4.6 Program Trace, Indirect Branch with Sync

If a Program Trace Synchronization message occurs on an instruction which transmits an indirect branch message, the Indirect Branch with Sync message is transmitted instead of the Program Trace Synchronization message. The Indirect Branch with Sync message contains the instruction count referring to the taken branch, as well as the complete PC value of the branch target.

The format for indirect branch messages with sync is shown below. The AVR32 OCD system never issues speculative branch messages and there is therefore no CANCEL packet.

Table 9-38. Indirect Branch message with Sync

Indirect Branch Message with Sync			Direction: From target
Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
32	F-ADDR	Variable	The full target address for a taken direct branch. Most significant bits that have a value of 0 may be truncated.
8	I-CNT	Variable	Number of bytes executed since the last taken branch.
2	EVT-ID	Fixed	Cause of indirect branch: 3: Exception entry 2: Call 1: Branch via register contents 0: Return
6	TCODE	Fixed	Value = 12

9.5.4.7 *Program Trace, Resource Full*

This message is output whenever an internal resource (sequential instruction counter) has reached its maximum value. To avoid losing information when this resource becomes full, the Resource Full message is transmitted. The information from this message is added with information from subsequent messages to interpret the full picture of what has transpired. Multiple Resource Full messages can occur before the arrival of the message that the information belongs with.

Table 9-39. Resource Full message

Program Trace, Resource Full			Direction: From target
Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	RDATA	Variable	Number of bytes executed since the last taken branch.
4	RCODE	Fixed	Resource Code. This code indicates which internal resource has reached its maximum value. Refer to Table 9-40 for details.
6	TCODE	Fixed	Value = 27

Table 9-40. Resource Code (RCODE) description

Resourc e Code	Resource	Data Packet Value
0b0000	Program Trace - Sequential Instruction Counter	Number of instruction units executed since the last taken branch.
0b0001 - 0b1111	Reserved	

9.5.4.8 Program Trace Correlation

Program Trace Correlation messages are used to correlate events to the program flow that may not be associated with the instruction stream (e.g. Data Trace Messages). The occurrence of an event listed in Table 9-41 will cause this message to be transmitted.

Table 9-41. Program Trace Correlation message

Program Trace Correlation			**Direction: From target**
Packet Size (bits)	**Packet Name**	**Packet Type**	**Description**
8	I-CNT	Variable	Number of instruction units executed since the last taken branch.
4	EVCODE	Fixed	Event Code. Refer to Table 9-42.
6	TCODE	Fixed	Value = 33

Table 9-42. Event Code (EVCODE) description

Event Code (EVCODE)	**Event Description**
0b0000	Entry into Debug Mode
0b0001	Entry into Low Power Mode
0b0010 - 0b0011	Reserved
0b0100	Program Trace Disabled
0b0101 - 0b1111	Reserved

9.5.5 Registers

Program trace is enabled using the TM field in the Development Control register.

9.6 Data Trace

9.6.1 Overview

The AVR32 OCD system provides data trace via the AUX port. The CPU data memory accesses can be monitored real-time using the Nexus class 2+ compliant Data Trace Unit. Both reads and writes can be traced.

Data Trace information is transmitted through data trace messages, which can be of read or write type, with or without sync. The messages contain information about the data address and value which triggered the trace. Data addresses can be complete (with sync), or compressed relative to the previous transmitted message (without sync). The value contains the data value read or written from the data cache, and is of the same width as the access size (byte, halfword, word, or doubleword).

The TM[1] bit in the Development Control register must be set to enable data trace. It is also possible to trigger data trace using watchpoints. In this case, TM[1] will be set or cleared automatically.

9.6.2 Using data trace channels as watchpoints

Data Trace is enabled for address ranges (trace channels) specified by pairs of Data Trace Start and End Address registers (DTSA/DTEA). Each data access within that boundary will generate an action as specified by the corresponding bits in the Data Trace Control register (DTC). The AVR32 OCD system currently supports two data trace channels.

While each channel can be used to trigger data trace messages, it is also possible to trigger watchpoint messages, providing flexibility when using the OCD system. Watchpoints can be ranged, i.e. trigger on all accesses between DTSA through DTEA, or trigger on a single location, if DTSA and DTEA are written to the same value.

Writing TnWP to one enables a watchpoint on accesses for data trace channel n. The watchpoint message is sent as a vendor defined trace watchpoint message.

It is possible to enable both trace and watchpoint on the same channel, but typically, only one of the options will be used.

9.6.3 Messages

The Trace Watchpoint Hit message is described in Section 9.4.5.2 on page 122.

9.6.3.1 Data Trace, Data Write (DTDW)

This message is output by the target processor when it detects a memory write that matches the OCD system's data trace attributes.

Table 9-43. Data Trace, Data Write message

Data Trace, Data Write message			**Direction: From target**
Packet Size	**Packet Name**	**Packet Type**	**Description**
8 / 16 / 32	DATA	Variable	The data value written. The size will vary depending on the load / store instruction being traced.
32	U-ADDR	Variable	The unique portion of the data write address, which is relative to the previous Data Trace Message (read or write).
2	DSZ	Fixed	Data size: 00 = 8 bits 01 = 16 bits 10 = 32 bits
6	TCODE	Fixed	Value=5

9.6.3.2 Data Trace, Data Write with Sync (DTDWS)

This message is an alternative to the Data Trace, Data Write Message. It is output instead of a Data Trace, Data Write Message whenever a memory write occurs that matches the debug logic's data trace attributes, and when one of the following conditions has occurred:

1. The processor has exited from reset. This synchronization message is required to allow the unique portion of the data write address of following Data Trace, Data Write Messages to be correctly interpreted by the tool.
2. When data trace is enabled during normal execution of the embedded processor.

3. Upon exit from a power-down state. This synchronization message is required to allow the unique portion of the data write address of following Data Trace, Data Write Messages to be correctly interpreted by the tool.
4. The Event-In pin has been asserted and a debug control register field specifies that EVTI pin action is to generate data trace synchronization.
5. An overrun condition had previously occurred in which one or more data trace occurrences were discarded by the target processor's debug logic. To inform the tool that an overrun condition occurred,the target outputs an Error Message (TCODE = 8) with an ECODE value of 00010 or 00111 immediately prior to the Data Trace, Data Write with Sync Message.
6. The Data Trace Message counter has expired indicating that at most 256 without-sync versions of Data Trace Messages have been sent since the last with-sync version.
7. A data write is detected following the processor exiting from Debug Mode.

Table 9-44. Data Trace, Data Write with Sync message

Data Trace, Data Write with Sync message			**Direction: From target**
Packet Size	**Packet Name**	**Packet Type**	**Description**
8 / 16 / 32	DATA	Variable	The data value written. The size will vary depending on the load / store instruction being traced.
32	F-ADDR	Variable	The full address of the memory location written. Most significant bits that have a value of 0 are truncated.
2	DSZ	Fixed	Data size: 00 = 8 bits 01 = 16 bits 10 = 32 bits
6	TCODE	Fixed	Value=13

9.6.3.3 *Data Trace, Data Read (DTDR)*

This message is output by the target processor when it detects a memory read that matches the OCD system's data trace attributes.

Table 9-45. Data Trace, Data Read message

Data Trace, Data Read message			**Direction: From target**
Packet Size	**Packet Name**	**Packet Type**	**Description**
8 / 16 / 32	DATA	Variable	The data value read. The size will vary depending on the load / store instruction being traced.
32	U-ADDR	Variable	The unique portion of the data read address, which is relative to the previous Data Trace Message (read or write).
2	DSZ	Fixed	Data size: 00 = 8 bits 01 = 16 bits 10 = 32 bits
6	TCODE	Fixed	Value=6

9.6.3.4 Data Trace, Data Read with Sync (DTDRS)

This message is an alternative to the Data Trace, Data Read Message. It is output instead of a Data Trace, Data Read Message whenever a memory read occurs that matches the debug logic's data trace attributes, and when one of the following conditions has occurred:

The processor has exited from reset. This synchronization message is required to allow the unique portion of the data write address of following Data Trace, Data Read Messages to be correctly interpreted by the tool.

When enabling data trace is during normal execution of the embedded processor.

Upon exit from a power-down state. This synchronization message is required to allow the unique portion of the data write address of following Data Trace, Data Read Messages to be correctly interpreted by the tool.

The Event-In pin has been asserted and a debug control register field specifies that EVTI pin action is to generate data trace synchronization.

An overrun condition had previously occurred in which one or more data trace occurrences were discarded by the target processor's debug logic. To inform the tool that an overrun condition occurred, the target outputs an Error Message (TCODE = 8) with an ECODE value of 00010 or 00111 immediately prior to the Data Trace, Data Read with Sync Message.

The periodic Data Trace Message counter has expired indicating that 255 without-sync versions of Data Trace Messages have been sent since the last with-sync version.

A data read is detected following the processor exiting from Debug Mode.

Table 9-46. Data Trace, Data Read with Sync message

Data Trace, Data Read with Sync message			**Direction: From target**
Packet Size	**Packet Name**	**Packet Type**	**Description**
8 / 16 / 32	DATA	Variable	The data value read. The size will vary depending on the load / store instruction being traced.
32	F-ADDR	Variable	The full address of the memory location written. Most significant bits that have a value of 0 are truncated.
2	DSZ	Fixed	Data size: 00 = 8 bits 01 = 16 bits 10 = 32 bits
6	TCODE	Fixed	Value=14

9.6.3.5 *Data Trace, Read-Modify-Write (DTRMW)*

This message is generated when a Read-Modify-Write (RMW) instruction is generated with a target address within an active data trace window. These instructions have the format "memc/s/t imm, bp", and can clear, set, or toggle a specified bit in memory.

Table 9-47. Data Trace, Read-Modify-Write message

Data Trace, Read-Modify-Write message			**Direction: From target**
Packet Size	**Packet Name**	**Packet Type**	**Description**
32	U-ADDR	Variable	The unique portion of the data write address, which is relative to the previous Data Trace Message (read or write).
5	BIT	Variable	Bit argument of the RMW instruction.
2	TYPE	Fixed	Bit operation: 00 = Reserved 01 = Clear 10 = Set 11 = Toggle
6	TCODE	Fixed	Value=58

9.6.3.6 *Data Trace, Read-Modify-Write with Sync (DTRMWS)*

This message is output instead of DTRMW under the same conditions as shown for DTDWS.

Table 9-48. Data Trace, Read-Modify-Write with Sync message

Data Trace, Read-Modify-Write with sync message			**Direction: From target**
Packet Size	**Packet Name**	**Packet Type**	**Description**
32	F-ADDR	Variable	The full address of the memory location written. Most significant bits that have a value of 0 are truncated.
5	BIT	Variable	Bit argument of the RMW instruction.
2	TYPE	Fixed	Bit operation: 00 = Reserved 01 = Clear 10 = Set 11 = Toggle
6	TCODE	Fixed	Value=59

9.6.4 Registers

9.6.4.1 Data Trace Control register (DTC)

This register controls actions taken on data accesses within all data trace channels.

Table 9-49. Data Trace Control Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	31:30	RWT0	0	RWT0 - Read/Write Trace channel 0 00 = No trace enabled x1 = Enable data read trace 1x = Enable data write trace
R/W	29:28	RWT1	0	RWT1 - Read/Write Trace channel 1 00 = No trace enabled x1 = Enable data read trace 1x = Enable data write trace
R	27:2	Reserved	0	
R/W	1	T1WP	0	T1WP - Trace Channel 1 Watchpoint
R/W	0	T0WP	0	T0WP - Trace Channel 0 Watchpoint

9.6.4.2 Data Trace Start/End Address register (DTSA/DTEA)

DTSAn and DTEAn define the inclusive data access range [DTSAn : DTEAn] for trace channel n. Each trace channel 0 and 1 has its own DTSA/DTEA register pair. If DTSA=DTEA, the trace channel will match on accesses to a single location. If DTSA>DTEA, no match will occur for the trace channel.

DTSA0, DTSA1

Table 9-50. Data Trace Start Address Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	31:0	DTSA	0	DTSA - Start address for trace visibility

DTEA0, DTEA1

Table 9-51. Data Trace End Address Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	31:0	DTEA	0	DTEA - End address for trace visibility

9.7 Ownership Trace

9.7.1 Functional description

The AVR32 OCD system implements Ownership Trace in compliance with the Nexus standard.

Ownership trace provides a macroscopic view, such as task flow reconstruction, when debugging software written in a high level (or object oriented) language. It offers the highest level of abstraction for tracking operating system software execution. This is especially useful when the developer is not interested in debugging at lower levels.

Ownership trace is especially important for embedded processors with a memory management unit, in which all processes can use the same virtual program and data spaces. Ownership trace offers development tools a mechanism to decipher which set of symbolics and sources are associated for lower levels of visibility and debugging.

Ownership trace information is transmitted out the AUX using an Ownership Trace Message. OTM facilitates ownership trace by providing visibility of which process ID or operating system task is activated. An Ownership Trace Message is transmitted to indicate when a new process/task is activated, allowing development tools to trace ownership flow. Additionally, an Ownership Trace Message is also transmitted periodically during runtime at a minimum frequency of every 256 Program Trace or Data Trace Messages.

In the AVR32, this feature is supported through an Ownership Trace Register, which automatically produces an Ownership Trace Message when written to. The RTOS scheduler routine writes the new process ID to this register during process switching using the *mtdr* instruction.

The TM[0] bit in the Development Control register must be set to enable ownership trace.

9.7.2 Messages

9.7.2.1 Ownership Trace (OT)

- The ownership trace message is sent:
- When the Ownership Trace Process ID (PID) register is written.
- When program trace with sync message is generated due to overflow in the periodic message counter.
- When a data trace with sync message is generated due to overflow in the periodic message counter.
- After a Transmit Queue overrun if the CPU has written to PID when the queue was full.

If there is no room in the Transmit Queue for the message, and the CPU is not halted to prevent overruns, an error message is produced.

Table 9-52. Ownership Trace Message

Ownership Trace Message			**Direction: From target**
Packet Size	**Packet Name**	**Packet Type**	**Description**
32	PROCESS	Fixed	Task / process ID.
6	TCODE	Fixed	Value = 2

9.7.3 Registers

9.7.3.1 Ownership Trace Process ID (PID)

The CPU should write the current Process ID value to this register, whenever the RTOS performs a process switch. This will automatically create an Ownership Trace Message to be transmitted to the tool. This register can be written from any privileged CPU mode.

The tool can read and write this register, although it is recommended that only the CPU writes this register.

Table 9-53. Ownership Trace Process ID (PID)

R/W	Bit Number	Field Name	Init. Val.	Description
RW	31:0	PROCESS	0	PROCESS - Process ID The unique Process ID number of the currently running process.

9.8 Memory Service Unit

The Memory Service Unit (MSU) provides access to complex memory operations, such as CRC checking and NanoTrace. The MSU is accessed by SAB registers, but these are not mapped into the OCD register space, and needs to be accessed with MEMORY_WORD_ACCESS or MEMORY_SERVICE_ACCESS JTAG commands. In addition the MSU registers are mapped in the system register space and are available tothe CPU. Refer to Section 2.5 ”System registers” on page 11 for details.

9.8.1 CRC

The MSU can calculate a Cyclic Redundancy Check (CRC) value for a memory area. The algorithm used is the industry standard CRC32 algorithm using the generator polynomial 0xEDB88320.

9.8.1.1 Starting CRC calculation

To calculate CRC for a memory range, you need to write the start address into the ADDRHI and ADDRLO registers, and the size of the memory range into the LENGTH register. Both the start address and the length must be word aligned.

The initial value used for the CRC calculation must be written to the DATA register. This value will usually be 0xFFFFFFFF, but can be e.g. the result of a previous CRC calculation if generating a common CRC of separate memory blocks.

Once completed, the calculated CRC value can be read out of the DATA register. The read value must be inverted to match standard CRC32 implementations, or kept non-inverted if used as starting point for subsequent CRC calculations.

If the device has enabled protection features, e.g. the protection fuse has been set on devices with onboard flash memory, it is only possible to calculate CRC on a predefined memory area. In most cases this area will be the entire onboard flash memory. The ADDRHI, ADDRLO, LENGTH, and DATA registers will be forced to predefined values once the CRC operation is started, and user-written values are ignored. This allows the user to verify the contents of a protected device, while denying malicious users the option of analyzing the memory contents through selective CRC calculations.

The actual test is started by writing OP_CRC to the CTRL register. A running CRC operation can be cancelled by writing OP_IDLE to CTRL.

9.8.1.2 Interpreting the results

The user should monitor the RESULT register. The possible values are:

Table 9-54. CRC results

Result	Description
NOT_IMPL	The CRC feature is not implemented in this device.
CANCELED	The CRC operation was canceled by writing a value different from OP_CRC to CTRL.
BUSY	The CRC operation is running.
DONE	The CRC operation has completed.
BUS_ERR	Part of the specified memory could not be read or written. The offending address can be read out of ADDRHI and ADDRLO.

9.8.2 NanoTrace

The MSU redirect OCD trace output from the normal trace output port to memory. This feature is called NanoTrace, and enables trace functionality with low cost debuggers, or even trace support in self hosted debuggers.

9.8.2.1 Starting NanoTrace

The memory range to write the trace data to is specified by writing the start address into the ADDRHI and ADDRLO registers, and the size of the memory range into the LENGTH register. In addition, the TAIL register may need to be updated, see below for details. Both the start address and the length must be word aligned.

The MSU starts expecting trace data when OP_NANOTRACE is written to the CTRL register. The OCD system must be separately configured to actually produce trace data.

The *ntbc* field of CTRL controls how the memory service behaves when the trace buffer is full.

9.8.2.2 Controlling buffer overflow

The trace buffer as specified by the initial ADDRHI, ADDRLO and LENGTH registers works as a circular buffer. The ADDRLO register is used by the MSU as the insert or head pointer, and the TAIL register is used by the debugger as the extract or tail pointer. ADDRHI is used together with both ADDRLO and TAIL to generate complete SAB addresses.

While writing trace data to memory, the ADDRLO and LENGTH registers are continuously updated with the current address being written and bytes remaining until the end of the buffer. When LENGTH reaches zero, the original contents of ADDRLO and LENGTH are restored, and trace data are again written from the beginning of the buffer.

The trace buffer is considered to be full when the ADDRLO register reaches the same value as TAIL. When starting trace, the TAIL register should be written to the same value as ADDRLO, meaning that the entire buffer can be written before ADDRLO matches TAIL. During tracing, a debugger may read out trace data from the buffer area between ADDRLO and TAIL, and then update TAIL. This will release space in the buffer, and can potentially allow continuos tracing if the trace buffer can be read out faster than trace data is generated.

When the buffer *does* fill up, the behavior of the MSU depends on the setting of the *ntbc* field:

- **OVER:** The unit keeps tracing, overwriting old trace data. The *wrap* field in the STATUS register is set to indicate that trace data are lost. The OCD system is asked to generate a synchronization message as soon as possible. The *wrap* field must be manually cleared by writing it to zero.

- **DISABLE:** The MSU leaves the NanoTrace mode, and goes to idle mode. The OCD system will continue as before, but trace data will go to the trace output port if enabled.
- **BREAK_STOP:** The MSU will ask the OCD system to enter debug mode, so no more trace data is generated. There will be a few more trace messages that do not fit in the buffer, so the CPU is forced to stop until the debugger clears room in the trace buffer. This ensures that no trace data is lost, but this also means that this mode will *deadlock* self hosted debuggers, as the CPU is stalled and can never release room in the trace buffer!
- **BREAK_FLUSH:** The MSU will ask the OCD system to enter debug mode, so no more trace data is generated. There will be a few more trace messages that do not fit in the buffer, but the MSU will flush these and allow the CPU to enter OCD mode. This means that the last few messages will be lost!

When restarting NanoTrace after entering debug mode using one of the BREAK modes, the TAIL register must be updated in order to tell the MSU that there is free space in the trace buffer. It is perfectly valid to update TAIL to the value it already contains - this is interpreted as making the entire buffer available. After this, the CPU can be restarted by issuing a *retd* instruction through the OCD system. If *ntbc* is set to BREAK_FLUSH the first new trace message will be a synchronization message.

9.8.2.3 Interpreting the results

The debugger should monitor the RESULT register. The possible values are:

Table 9-55. NanoTrace results

Result	Description
NOT_IMPL	The NanoTrace feature is not implemented in this device.
CANCELED	The NanoTrace operation was canceled by writing a value different from OP_NANOTRACE to CTRL.
BUSY	The NanoTrace operation is running.
DONE	The NanoTrace operation has completed. This can only happen when the *ntbc* field is written to DISABLE.
BUS_ERR	Part of the specified memory could not be written to. The offending address can be read out of ADDRHI and ADDRLO.

9.8.3 MSU Register summary

Table 9-56. MSU Register Summary

Offset	Register	Name	Access	Reset State
0	Address register, high part	ADDRHI	Read/Write	-
1	Address register, low part	ADDRLO	Read/Write	-
2	Length register	LENGTH	Read/Write	-
3	Control register	CTRL	Read/Write	0x0
4	Status register	STATUS	Read-only	0x0
5	Data register	DATA	Read/Write	0x0
6	Tail address register	TAIL	Read/Write	-

9.8.3.1 Address Register, High Part

Name: ADDRHI

Access Type: Read/Write

Address offset: 0

31	30	29	28	27	26	25	24
–	–	–	–	–	–	–	–
23	22	21	20	19	18	17	16
–	–	–	–	–	–	–	–
15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–
7	6	5	4	3	2	1	0
–	–	–	–	ADDRHI			

- **ADDRHI: *Address High part***

Bits 35:32 of full SAB address

9.8.3.2 Address Register, Low Part

Name: ADDRLO

Access Type: Read/Write

Address offset: 1

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td></tr>
<tr><td colspan="8">ADDRLO</td></tr>
<tr><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr>
<tr><td colspan="8">ADDRLO</td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td></tr>
<tr><td colspan="8">ADDRLO</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="6">ADDRLO</td><td>0</td><td>0</td></tr>
</table>

- **ADDRLO: Address Low part**

Bits 31-2 of full SAB address.

- **Bits 1:0**

Always zero.

9.8.3.3 Length Register

Name: LENGTH

Access Type: Read/Write

Address offset: 2

31	30	29	28	27	26	25	24
LENGTH							
23	22	21	20	19	18	17	16
LENGTH							
15	14	13	12	11	10	9	8
LENGTH							
7	6	5	4	3	2	1	0
LENGTH						0	0

- **LENGTH: Length value**

- **Bits 1:0**

Always zero.

9.8.3.4 Control Register

Name: CTRL

Access Type: Read/Write

Address offset: 3

31	30	29	28	27	26	25	24
–	–	–	–	–	–	–	–
23	22	21	20	19	18	17	16
–	–	–	–	–	–	–	–
15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–
7	6	5	4	3	2	1	0
–	–	NTBC		OP			

- **OP: Requested operation:**

Table 9-57. OP field of control register

OP	Name	Description
0	NONE	No operation, or cancel current operation
1	CRC	Calculate CRC of memory area
2	NTRACE	Write trace data from debug system to memory
Others	N/A	Reserved

- **NTBC: NanoTrace Buffer Control**

What to do when trace buffer is full:

Table 9-58. NTBC field of control register

OP	Name	Description
0	OVER	Overwrite old buffer contents
1	DISABLE	Disable nanotrace
2	BREAK_STOP	Make the CPU enter debug mode, but stop CPU until place has been freed for the last few trace frames. *Note:* This may deadlock self-hosted debuggers!
3	BREAK_FLUSH	Make the CPU enter debug mode, and flush the last few trace frames that don’t fit in the buffer.

9.8.3.5 Status Register

Name: STATUS

Access Type: Read/Write

Address offset: 4

31	30	29	28	27	26	25	24
–	–	–	–	–	–	–	–
23	22	21	20	19	18	17	16
–	–	–	–	–	–	–	–
15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–
7	6	5	4	3	2	1	0
–	–	–	–	WRAP	RESULT		

- **RESULT: Result of current or last operation**

Table 9-59. Result field of status register

OP	Name	Description
0	DONE	The previous operation finished successfully
1	BUSY	Some operation is currently active
2	NOT_IMPL	The requested operation is not implemented
3	BUS_ERR	Unable to access part of the requested memory area
4	FAILED	The requested operation failed
5	LOCKED	The requested operation cannot be performed because chip security features are enabled
6	CANCELED	The previous operation was canceled by the user
Others	N/A	Reserved

- **WRAP**

The NanoTrace write buffer address has wrapped back to the start address.

9.8.3.6 Data Register

Name: DATA

Access Type: Read/Write

Address offset: 5

31	30	29	28	27	26	25	24
DATA							
23	22	21	20	19	18	17	16
DATA							
15	14	13	12	11	10	9	8
DATA							
7	6	5	4	3	2	1	0
DATA							

- **DATA: Generic Data Register**

9.8.3.7 Tail Address Register

Name: TAIL

Access Type: Read/Write

Address offset: 7

31	30	29	28	27	26	25	24
TAIL							
23	22	21	20	19	18	17	16
TAIL							
15	14	13	12	11	10	9	8
TAIL							
7	6	5	4	3	2	1	0
TAIL						0	0

- **TAIL: Tail Address Register for NanoTrace buffer.**

- **Bits 1:0**

Always zero.

9.9 OCD Message Summary

Table 9-60. Message Summary

TCODE	Message	Public / Vendor Defined	Page
0	Debug Status (DEBS)	Public	page 100
1	Reserved		
2	Ownership Trace (OT)	Public	page 137
3	Program Trace, Direct Branch (PTDB)	Public	page 127
4	Program Trace, Indirect Branch (PTIB)	Public	page 127
5	Data Trace, Data Write (DTDW)	Public	page 132
6	Data Trace, Data Read (DTDR)	Public	page 133
7	Reserved		
8	Error (ERROR)	Public	page 116
9	Program Trace Synchronization (PTSY)	Public	page 128
10	Reserved		
11	Program Trace, Direct Branch with Sync (PTDBS)	Public	page 129
12	Program Trace, Indirect Branch with Sync (PTIBS)	Public	page 129
13	Data Trace, Data Write with Sync (DTDWS)	Public	page 132
14	Data Trace, Data Read with Sync (DTDRS)	Public	page 134
15	Watchpoint Hit (WH)	Public	page 122
16–26	Reserved		
27	Program Trace Resource Full (PTRF)	Public	page 130
28–32	Reserved		
33	Program Trace Correlation (PTC)	Public	page 131
34–55	Reserved		
56	Trace Watchpoint Hit (TWH)	Vendor	page 122
57	Direct Branch with Target Address (DBTA)	Vendor	page 127
58	Data Trace, Read-Modify-Write (DTRMW)	Vendor	page 135
59	Data Trace, Read-Modify-Write with Sync (DTRMWS)	Vendor	page 135
60-62	Reserved	Vendor	
63 (0x3F)	Vendor Defined Extension Message Reserved	Vendor	

Table 9-62 shows the messages which can be transmitted by the target on the AUX port. OCD registers can be written by the tool using the JTAG mechanism described in "Debug Port" on page 109.

Table 9-63 shows the format of the transmitted messages. Packets shown in bold are variable length, the others are fixed length. All variable length packets can be truncated by omitting leading zeroes, but will always end on a port boundary.

Table 9-61. Message formats

Nexus Message	**Message format** **TCODE [5:0]**	**Packet 1**	**Packet 2**	**Packet 3**
Debug Status	0	STATUS[31:0]		
Ownership Trace	2	PROCESS [31:0]	-	-
Error	8	ECODE[4:0]	-	-
Program Trace, Direct Branch	3	**I-CNT[7:0]**	-	-
Program Trace, Direct Branch with Target Address	57	**I-CNT[7:0]**	**U-ADDR[31:0]**	-
Program Trace, Indirect Branch	4	EVT-ID[1:0]	**I-CNT[7:0]**	**U-ADDR[31:0]**
Program Trace Synchronization	9	**I-CNT[7:0]**	**PC[31:0]**	-
Program Trace, Direct Branch with Sync	11	**I-CNT[7:0]**	**F-ADDR[31:0]**	-
Program Trace, Indirect Branch with Sync	12	EVT-ID[1:0]	**I-CNT[7:0]**	**F-ADDR[31:0]**
Program Trace Resource Full	27	RCODE[3:0]	**RDATA[7:0]**	
Program Trace Correlation	33	EVCODE[3:0]	**I-CNT[7:0]**	
Data Trace, Data Write	5	DSZ[1:0]	**U-ADDR[31:0]**	**DATA[31:0]**
Data Trace, Data Read	6	DSZ[1:0]	**U-ADDR[31:0]**	**DATA[31:0]**
Data Trace, Read-Modify-Write	58	TYPE[1:0]	BIT[4:0]	**U_ADDR[31:0]**
Data Trace, Data Write with Sync	13	DSZ[1:0]	**F-ADDR[31:0]**	**DATA[31:0]**
Data Trace, Data Read with Sync	14	DSZ[1:0]	**F-ADDR[31:0]**	**DATA[31:0]**
Data Trace, Read-Modify-Write with Sync	59	TYPE[1:0]	BIT[4:0]	**F_ADDR[31:0]**
Watchpoint Hit	15	WPHIT[7:0]	-	-
Trace Watchpoint Hit	56	WPHIT[1:0]	-	-

9.10 OCD Message Summary

Table 9-62. Message Summary

TCODE	Message	Public / Vendor Defined	Page
0	Debug Status (DEBS)	Public	page 100
1	Reserved		
2	Ownership Trace (OT)	Public	page 137
3	Program Trace, Direct Branch (PTDB)	Public	page 127
4	Program Trace, Indirect Branch (PTIB)	Public	page 127
5	Data Trace, Data Write (DTDW)	Public	page 132
6	Data Trace, Data Read (DTDR)	Public	page 133
7	Reserved		
8	Error (ERROR)	Public	page 116
9	Program Trace Synchronization (PTSY)	Public	page 128
10	Reserved		
11	Program Trace, Direct Branch with Sync (PTDBS)	Public	page 129
12	Program Trace, Indirect Branch with Sync (PTIBS)	Public	page 129
13	Data Trace, Data Write with Sync (DTDWS)	Public	page 132
14	Data Trace, Data Read with Sync (DTDRS)	Public	page 134
15	Watchpoint Hit (WH)	Public	page 122
16–26	Reserved		
27	Program Trace Resource Full (PTRF)	Public	page 130
28–32	Reserved		
33	Program Trace Correlation (PTC)	Public	page 131
34–55	Reserved		
56	Trace Watchpoint Hit (TWH)	Vendor	page 122
57	Direct Branch with Target Address (DBTA)	Vendor	page 127
58	Data Trace, Read-Modify-Write (DTRMW)	Vendor	page 135
59	Data Trace, Read-Modify-Write with Sync (DTRMWS)	Vendor	page 135
60-62	Reserved	Vendor	
63 (0x3F)	Vendor Defined Extension Message Reserved	Vendor	

Table 9-62 shows the messages which can be transmitted by the target on the AUX port. OCD registers can be written by the tool using the JTAG mechanism described in "Debug Port" on page 109.

Table 9-63 shows the format of the transmitted messages. Packets shown in bold are variable length, the others are fixed length. All variable length packets can be truncated by omitting leading zeroes, but will always end on a port boundary.

Table 9-63. Message formats

Nexus Message	**Message format**			
	TCODE [5:0]	**Packet 1**	**Packet 2**	**Packet 3**
Debug Status	0	STATUS[31:0]		
Ownership Trace	2	PROCESS [31:0]	-	-
Error	8	ECODE[4:0]	-	-
Program Trace, Direct Branch	3	**I-CNT[7:0]**	-	-
Program Trace, Direct Branch with Target Address	57	**I-CNT[7:0]**	**U-ADDR[31:0]**	-
Program Trace, Indirect Branch	4	EVT-ID[1:0]	**I-CNT[7:0]**	**U-ADDR[31:0]**
Program Trace Synchronization	9	**I-CNT[7:0]**	**PC[31:0]**	-
Program Trace, Direct Branch with Sync	11	**I-CNT[7:0]**	**F-ADDR[31:0]**	-
Program Trace, Indirect Branch with Sync	12	EVT-ID[1:0]	**I-CNT[7:0]**	**F-ADDR[31:0]**
Program Trace Resource Full	27	RCODE[3:0]	**RDATA[7:0]**	
Program Trace Correlation	33	EVCODE[3:0]	**I-CNT[7:0]**	
Data Trace, Data Write	5	DSZ[1:0]	**U-ADDR[31:0]**	**DATA[31:0]**
Data Trace, Data Read	6	DSZ[1:0]	**U-ADDR[31:0]**	**DATA[31:0]**
Data Trace, Read-Modify-Write	58	TYPE[1:0]	BIT[4:0]	**U_ADDR[31:0]**
Data Trace, Data Write with Sync	13	DSZ[1:0]	**F-ADDR[31:0]**	**DATA[31:0]**
Data Trace, Data Read with Sync	14	DSZ[1:0]	**F-ADDR[31:0]**	**DATA[31:0]**
Data Trace, Read-Modify-Write with Sync	59	TYPE[1:0]	BIT[4:0]	**F_ADDR[31:0]**
Watchpoint Hit	15	WPHIT[7:0]	-	-
Trace Watchpoint Hit	56	WPHIT[1:0]	-	-

9.11 OCD Register Summary

Use the index shown in the "Register index" column when accessing OCD registers by the Nexus access mechanism (see Section 9.3.2 on page 109).Use the index shown in the "mtdr/mfdr index" column when accessing OCD registers by *mtdr/mfdr* instructions from the CPU (see Section 9.2.10 on page 98). These indexes are identical to the register index multiplied by 4.

Table 9-64. OCD Register Summary

Register Index	mtdr/mfdr index	Register	Access Type	Page
0	0	Device ID (DID)	R	page 100
1	4	Reserved	—	
2	8	Development Control (DC)	R/W	page 104
3	12	Reserved	—	
4	16	Development Status (DS)	R	page 106
5-6	20-24	Reserved	—	
7	28	Reserved	—	
8	32	Reserved	—	
9	36	Reserved	—	
10	40	Reserved	—	
11	44	Watchpoint Trigger (WT)	R/W	page 125
12	48	Reserved	—	
13	52	Data Trace Control (DTC)	R/W	page 136
14–15	56-60	Data Trace Start Address (DTSA) Channel 0 to 1	R/W	page 136
16-17	64-68	Reserved	—	
18–19	72-76	Data Trace End Address (DTEA) Channel 0 to 1	R/W	page 136
20-21	80-84	Reserved	—	
22	88	PC Breakpoint/Watchpoint Control 0A (BWC0A)	R/W	page 123
23	92	PC Breakpoint/Watchpoint Control 0B (BWC0B)	R/W	page 123
24	96	PC Breakpoint/Watchpoint Control 1A (BWC1A)	R/W	page 123
25	100	PC Breakpoint/Watchpoint Control 1B (BWC1B)	R/W	page 123
26	104	PC Breakpoint/Watchpoint Control 2A (BWC2A)	R/W	page 123
27	108	PC Breakpoint/Watchpoint Control 2B (BWC2B)	R/W	page 123
28	112	Data Breakpoint/Watchpoint Control 3A (BWC3A)	R/W	page 124
29	116	Data Breakpoint/Watchpoint Control 3B (BWC3B)	R/W	page 124
30	120	PC Breakpoint/Watchpoint Address 0A (BWA0A)	R/W	page 122
31	124	PC Breakpoint/Watchpoint Address 0B (BWA0B)	R/W	page 122
32	128	PC Breakpoint/Watchpoint Address 1A (BWA1A)	R/W	page 122
33	132	PC Breakpoint/Watchpoint Address 1B (BWA1B)	R/W	page 122

Table 9-64. OCD Register Summary

Register Index	mtdr/mfdr index	Register	Access Type	Page
34	136	PC Breakpoint/Watchpoint Address 2A (BWA2A)	R/W	page 122
35	140	PC Breakpoint/Watchpoint Address 2B (BWA2B)	R/W	page 122
36	144	Data Breakpoint/Watchpoint Address 3A (BWA3A)	R/W	page 123
37	148	Data Breakpoint/Watchpoint Address 3B (BWA3B)	R/W	page 123
38	152	Breakpoint/Watchpoint Data 3A (BWD3A)	R/W	page 123
39	156	Breakpoint/Watchpoint Data 3B (BWD3B)	R/W	page 123
40–65	160-260	Reserved	—	
64	256	Nexus Configuration (NXCFG)	R	page 100
65	260	Debug Instruction Register (DINST)	R/W	page 108
66	264	Debug Program Counter (DPC)	R/W	page 109
67	268	Reserved	—	
68	272	Debug Communication CPU Register (DCCPU)	R/W	page 101
69	276	Debug Communication Emulator Register (DCEMU)	R/W	page 102
70	280	Debug Communication Status Register (DCSR)	R/W	page 102
71	284	Ownership Trace Process ID (PID)	R/W	page 137
72	288	Debug Communication Control Register (DCCR)	R/W	page 103
73	292	Peripheral Debug Register(PDBG)	R/W	
74-75	296-300	Reserved	—	
76	304	AUX port Control (AXC)	R/W	page 116
77– 255	308-1020	Reserved	—	

10. Revision History

Doc. Rev.	Date	Comments
32002F	2010-03-12	Improved description of events and priority. Replaced invalid reference in the OCD.PDBG register. Note added about overall system interrupt latency. Added MSU system registers.
32002E	2009-09-01	Added Floating-Point hardware description.
32002D	2009-08-01	Added OCD DCCPU and DCEMU interrupts. Added PDBG register for individual module masks. Added AVR32 architecture revision 3 secure state support. COUNT/COMPARE system register reset-on-match now programmable by CPUCR Corrected LDM, STM, and SCALL instruction cycle count in cycle count chapter. Corrected maximum IRQ latency in the Pipeline chapter.
32002C	2007-11-19	MPU compilant with revision 2 of AVR32 Architecture. Added cycle counts for new instruction in version 2 of the CPU. Added COUNT/COMPARE system register reset-on-match. Added CPU Local Bus. Reconfigured OCD AXC register.
32002B	2007-08-03	Added Memory Service Unit (MSU) description. Added description of peripheral behavior in Debug.
32002A	2007-03-30	Initial revision.

Table of contents

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
avr32@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Requests
www.atmel.com/literature

Disclaimer: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. **EXCEPT AS SET FORTH IN ATMEL'S TERMS AND CONDITIONS OF SALE LOCATED ON ATMEL'S WEB SITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.** Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel's products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

Microchip:

AT32UC3A3256S-ALUR AT32UC3A3256S-CTUR AT32UC3A364S-ALUR AT32UC3A364S-ALUT AT32UC3A364S-CTUR AT32UC3A364S-CTUT