BDTIC www.bdtic.com/Semiconductor

32-bit AVR® Microcontroller

AVR32 AP Technical Reference Manual

Feature Summary

- **32-bit load/store AVR32B 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**
- **Shadowed interrupt context for INT3 and multiple interrupt priority levels**
- **Privileged and unprivileged modes enabling efficient and secure Operating Systems**
- **Full MMU allows for operating systems with memory protection**
- **Instruction and data caches**
- **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**
- **SIMD extention for media applications**
- **Dynamic branch prediction and return address stack for fast change-of-flow**
- **Powerful On-Chip Debug system**
- **Coprocessor interface**

32001A–AVR32–06/06

www.BDTIC.com/ATMEL

1. Introduction

AVR®32 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.

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 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 have 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.

www.BDTIC.com/ATMEL

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 architecture defines several microarchitectures 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 Event handling

The AVR32 incorporates a powerful event handling scheme. The different event sources, like "Illegal opcode" and external interrupt requests, have different priority levels, ensuring a well-defined behavior 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. 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

The AVR32 architecture defines a Java® hardware acceleration option, in the form of a Java Virtual Machine hardware implementation.

www.BDTIC.com/ATMEL

1.5 Microarchitectures

The AVR32 architecture defines different microarchitectures. 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 following microarchitectures are defined:

1.5.1 AVR32A

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.

1.5.2 AVR32B

The AVR32B microarchitecture is targeted at applications where interrupt latency is important. The AVR32B therefore implements dedicated registers to hold the status register and return address for interrupts, exceptions and supervisor calls. This information does not need to be written to the stack, and latency is therefore reduced. Additionally, AVR32B allows hardware shadowing of the registers in the register file. The INT0 to INT3 contexts may have dedicated versions of the registers in the register file, allowing the interrupt routine to start executing immediately.

The *scall*, *rete* and *rets* instructions use the dedicated status register and return address registers in their operation. No stack accesses are performed.

www.BDTIC.com/ATMEL

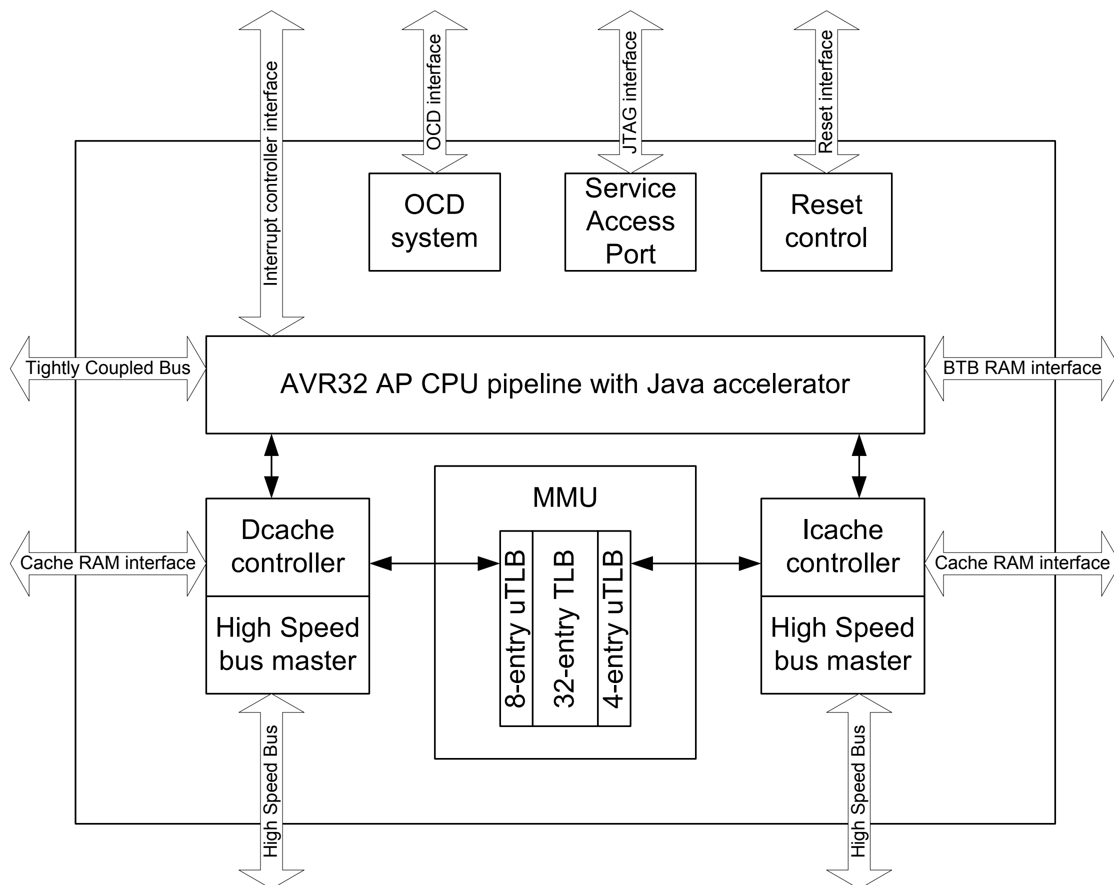
1.6 The AVR32 AP implementation

The first implementation of the AVR32B microarchitecture is designed as an application processor and called AVR32 AP. This implementation targets high-performance applications in the DSP, multimedia and wireless segment, and provides:

- Advanced OCD system.
- Efficient data and instruction caches.
- Full MMU.
- Java acceleration is implemented in hardware.
- Fast interrupt handling is provided through shadowed register banks for interrupt priority 3.
- SIMD extension.
- DSP extension.
- Service Access Port (SAP) that gives an external JTAG controller access to memories and registers inside the AVR32 AP core.

Figure 1-1 on page 5 displays the contents of AVR32 AP:

Figure 1-1. Overview of AVR32 AP.



www.BDTIC.com/ATMEL

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 AVR32 AP.

2.1 Architectural compatibility

AVR32 AP is fully compatible with the Atmel AVR32B architecture.

2.2 Implementation options

2.2.1 Memory management

AVR32 AP implements a full MMU as specified by the AVR32 architecture.

2.2.2 Java support

AVR32 AP implements a Java Extention Module (JEM) as defined in the AVR32 architecture.

2.3 Register file configuration

The AVR32B architecture specifies that the exception contexts may have a different number of shadowed registers in different implementations. The following shadow model is used in AVR32 AP.

Figure 2-1. Register file configuration. Shadowed registers are marked in grey.

Application	Supervisor	INT0	INT1	INT2	INT3	Exception	NMI
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
LR	LR	LR	LR	LR	LR_INT3	LR	LR
SP_APP	SP_SYS	SP_SYS	SP_SYS	SP_SYS	SP_SYS	SP_SYS	SP_SYS
R12	R12	R12	R12	R12	R12_INT3	R12	R12
R11	R11	R11	R11	R11	R11_INT3	R11	R11
R10	R10	R10	R10	R10	R10_INT3	R10	R10
R9	R9	R9	R9	R9	R9_INT3	R9	R9
R8	R8	R8	R8	R8	R8_INT3	R8	R8
R7	R7	R7	R7	R7	R7	R7	R7
R6	R6	R6	R6	R6	R6	R6	R6
R5	R5	R5	R5	R5	R5	R5	R5
R4	R4	R4	R4	R4	R4	R4	R4
R3	R3	R3	R3	R3	R3	R3	R3
R2	R2	R2	R2	R2	R2	R2	R2
R1	R1	R1	R1	R1	R1	R1	R1
R0	R0	R0	R0	R0	R0	R0	R0
SR	SR	SR	SR	SR	SR	SR	SR
	RSR_SUP	RSR_INT0	RSR_INT1	RSR_INT2	RSR_INT3	RSR_EX	RSR_NMI
	RAR_SUP	RAR_INT0	RAR_INT1	RAR_INT2	RAR_INT3	RAR_EX	RAR_NMI

www.BDTIC.com/ATMEL

2.4 Status register configuration

The Status Register (SR) is splitted into two halfwords, one upper and one lower. The lower word contains the C, Z, N, V and Q condition code flags and the R, T and L bits, while the upper halfword contains information about the mode and state the processor executes in.

Figure 2-2. The Status Register high halfword.

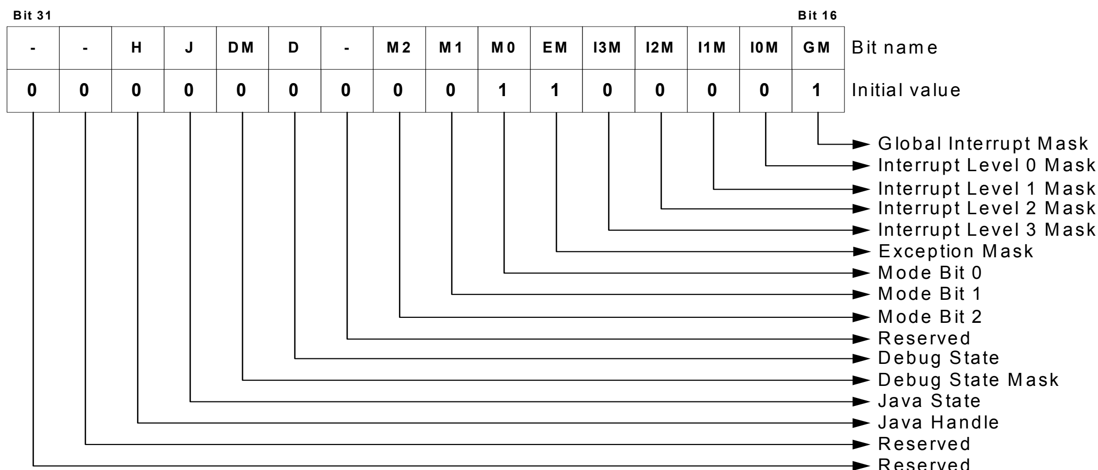
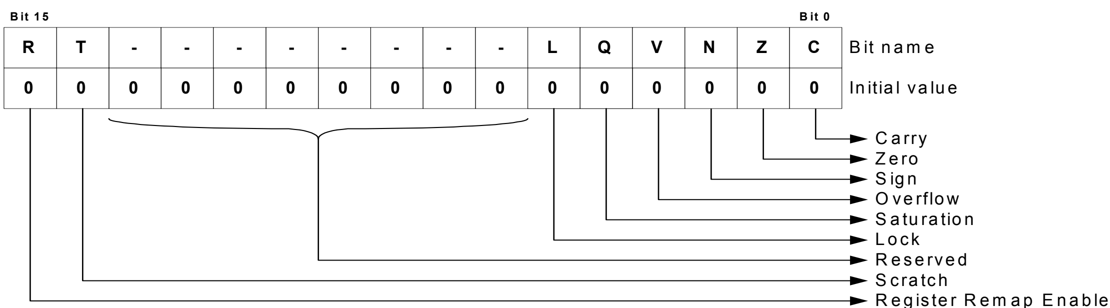


Figure 2-3. The Status Register low halfword.



H - Java Handle

This bit is included to support different heap types in the Java Virtual Machine. For more details, see the Java Technical Reference manual. The bit is cleared at reset.

J - Java state

The processor is in Java state when this bit is set. The incoming instruction stream will be decoded as a stream of Java bytecodes, not RISC opcodes. The bit is cleared at reset. This bit should not be modified by the user as undefined behaviour may result.

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.

www.BDTIC.com/ATMEL

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 manually.

M2, M1, M0 - Execution Mode

These bits show the active execution mode. The different settings for the different modes are shown in Table 2-1. 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, or execution of certain instructions like *scall*, *rets* and *rete*. Undefined behaviour may result if the user tries to modify these bits manually.

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.

www.BDTIC.com/ATMEL

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.

R - Java Register Remap

When this bit is set, the addresses of the registers in the register file is dynamically changed. This allows efficient use of the register file registers as a stack. For more details, see the Java Technical Reference Manual. The R bit is cleared at reset. Undefined behaviour may result if this bit is modified by the user.

T - Scratch bit

Not used by any instruction, but can be manipulated by application software as a scratchpad bit. This bit is cleared after 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.

www.BDTIC.com/ATMEL

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 AVR32 AP. It also identifies their address and the pipeline stage in which it is located. The programmer is responsible for maintaining correct sequencing of any instructions following a *mtsr* instruction.

Table 2-2. System Registers implemented in AVR32 AP

Reg #	Address	Name	Function	Location in pipeline
0	0	SR	Status Register	A1
1	4	EVBA	Exception Vector Base Address	A1
2	8	ACBA	Application Call Base Address	A1
3	12	CPUCR	CPU Control Register	A1
4	16	ECR	Exception Cause Register	A1
5	20	RSR_SUP	Return Status Register for supervisor context	A1
6	24	RSR_INT0	Return Status Register for INT 0 context	A1
7	28	RSR_INT1	Return Status Register for INT 1 context	A1
8	32	RSR_INT2	Return Status Register for INT 2 context	A1
9	36	RSR_INT3	Return Status Register for INT 3 context	A1
10	40	RSR_EX	Return Status Register for Exception context	A1
11	44	RSR_NMI	Return Status Register for NMI context	A1
12	48	RSR_DBG	Return Status Register for Debug Mode	A1
13	52	RAR_SUP	Return Address Register for supervisor context	A1
14	56	RAR_INT0	Return Address Register for INT 0 context	A1
15	60	RAR_INT1	Return Address Register for INT 1 context	A1
16	64	RAR_INT2	Return Address Register for INT 2 context	A1
17	68	RAR_INT3	Return Address Register for INT 3 context	A1
18	72	RAR_EX	Return Address Register for Exception context	A1
19	76	RAR_NMI	Return Address Register for NMI context	A1
20	80	RAR_DBG	Return Address Register for Debug Mode	A1
21	84	JECR	Java Exception Cause Register	A1
22	88	JOSP	Java Operand Stack Pointer	ID
23	92	JAVA_LV0	Java Local Variable 0	A1
24	96	JAVA_LV1	Java Local Variable 1	A1
25	100	JAVA_LV2	Java Local Variable 2	A1
26	104	JAVA_LV3	Java Local Variable 3	A1
27	108	JAVA_LV4	Java Local Variable 4	A1
28	112	JAVA_LV5	Java Local Variable 5	A1

www.BDTIC.com/ATMEL

Table 2-2. System Registers implemented in AVR32 AP (Continued)

Reg #	Address	Name	Function	Location in pipeline
29	116	JAVA_LV6	Java Local Variable 6	A1
30	120	JAVA_LV7	Java Local Variable 7	A1
31	124	JTBA	Java Trap Base Address	A1
32	128	JBCR	Java Write Barrier Control Register	A1
64	256	CONFIG0	Configuration register 0	TCB
65	260	CONFIG1	Configuration register 1	TCB
66	264	COUNT	Cycle Counter register	TCB
67	268	COMPARE	Compare register	TCB
68	272	TLBEHI	TLB Entry High	TCB
69	276	TLBELO	TLB Entry Low	TCB
70	280	PTBR	Page Table Base Register	TCB
71	284	TLBEAR	TLB Exception Address Register	TCB
72	288	MMUCR	MMU Control Register	TCB
73	292	TLBARLO	TLB Accessed Register Low	TCB
74	296	TLBARHI	TLB Accessed Register High	TCB
75	300	PCCNT	Performance Clock Counter	TCB
76	304	PCNT0	Performance Counter 0	TCB
77	308	PCNT1	Performance Counter 1	TCB
78	312	PCCR	Performance Counter Control Register	TCB
79	316	BEAR	Bus Error Address Register	TCB
192	768	SABAL	SAB Address Low Register	TCB
193	772	SABAH	SAB Address High Register	TCB
194	776	SABD	SAB Data Register	TCB

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 on page 7.

EVBA - Exception Vector Base Address

This register contains a pointer to the exception routines. All exception routines starts at this address, or at a defined offset relative to the address. Special alignment requirements apply for EVBA, see Section 3.10 ”Event handling” on page 30.

ACBA - Application Call Base Address

Pointer to the start of a table of function pointers. Subroutines residing in this space can be called by the compact *acall* instruction. This facilitates efficient reuse of code. Keeping this base pointer as a register facilitates multiple application spaces. ACBA is a full 32 bit register, but the

www.BDTIC.com/ATMEL

lowest bit should be written to zero, making ACBA halfword 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

Bit	Name	Reset	Access	Description
31 - 24	COP7EN - COP0EN	0	Read/write	Enable bit for coprocessor 7 to coprocessor 0. The corresponding coprocessor is enabled if this bit is written to one by software. Can be written to one only if the corresponding coprocessor is present in the system. Attempting to issue a coprocessor instruction to a coprocessor whose enable bit is cleared, will result in a coprocessor absent exception.
5	IEE	1	Read/write	Imprecise Execution Enable. Required for various OCD features, see Section 9. "OCD system" on page 86. If cleared, memory operations will require several additional clock cycles.
4	IBE	1	Read/write	Imprecise Breakpoint Enable. Required for various OCD features, see Section 9. "OCD system" on page 86. If cleared, memory operations will require an additional clock cycle.
3	RE	1	Read/write	If set, the return stack is enabled. Disabling the return stack will empty it, removing all entries.
2	FE	1	Read/write	If set, branch instructions can be folded with other instructions.
1	BE	1	Read/write	If set, branch prediction is enabled.
0	BI	-	Read-0/write-1	BTB invalidate. Writing to 1 will invalidate all entries in the BTB.
Other	-	-	Read-0/write-0	Unused. Read as 0. Should be written as 0.

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 a 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_SUP, RSR_INT0, RSR_INT1, RSR_INT2, RSR_INT3, RSR_EX, RSR_NMI - Return Status Registers

If a request for a mode change like an interrupt request is accepted when executing in a context *C*, the Status Register values in context *C* are automatically stored in the Return Status Register (RSR) associated with the interrupt context *I*. When the execution in the interrupt state *I* is fin-

www.BDTIC.com/ATMEL

ished and the *rets / rete* instruction is encountered, the RSR associated with *I* is copied to SR, and the execution continues in the original context *C*.

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_SUP, RAR_INT0, RAR_INT1, RAR_INT2, RAR_INT3, RAR_EX, RAR_NMI - Return Address Registers

If a request for a mode change, for instance an interrupt request, is accepted when executing in a context *C*, the re-entry address of context *C* is automatically stored in the Return Address Register (RAR) associated with the interrupt context *I*. When the execution in the interrupt state *I* is finished and the *rets / rete* instruction is encountered, a change-of-flow to the address in the RAR associated with *I*, and the execution continues in the original context *C*.

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.

JECR - Java Exception Cause Register

This register contains information needed for Java traps. See Java Technical Reference Manual for details.

JOSP - Java Operand Stack Pointer

This register holds the Java Operand Stack Pointer. See Java Technical Reference Manual for details. The register is initialized to 0 at reset.

JAVA_LVx - Java Local Variable Registers

The Java Extension Module uses these registers to temporarily store local variables. See Java Technical Reference Manual for details.

JTBA - Java Trap Base Address

This register contains the base address to the program code for the trapped Java instructions. See Java Technical Reference Manual for details.

JBCR - Java Write Barrier Control Register

This register is used by the garbage collector in the Java Virtual Machine. See Java Technical Reference Manual for details.

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 2.6 on page 16.

COUNT - Cycle Counter Register

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 interrupt functionality. The COUNT

www.BDTIC.com/ATMEL

register is written to zero upon reset. Incrementation of the COUNT register can not be disabled. The COUNT register will increment even though a compare interrupt is pending.

COMPARE - Cycle Counter Compare Register

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. This 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.

TLBEHI - MMU TLB Entry Register High Part

Used to interface the CPU to the TLB. The contents and functionality of the register is described in detail in Section 4. on page 48.

TLBELO - MMU TLB Entry Register Low Part

Used to interface the CPU to the TLB. The contents and functionality of the register is described in detail in Section 4. on page 48.

PTBR - MMU Page Table Base Register

Contains a pointer to the start of the Page Table. The contents and functionality of the register is described in detail in Section 4. on page 48.

TLBEAR - MMU TLB Exception Address Register

Contains the virtual address that caused the most recent MMU error. The contents and functionality of the register is described in detail in Section 4. on page 48.

MMUCR - MMU Control Register

Used to control the MMU and the TLB. The contents and functionality of the register is described in detail in Section 4. on page 48.

TLBARLO/HI - MMU TLB Accessed Register Low/High

Contains the Accessed bits for the TLB. The contents and functionality of the register is described in detail in Section 4. on page 48.

PCCNT - Performance Clock Counter

Clock cycle counter for performance counters. The contents and functionality of the register is described in detail in the AVR32 Architecture Manual.

PCNT0 / PCNT1 - Performance Counter 0 / 1

Counts the events specified by the Performance Counter Control Register. The contents and functionality of the register is described in detail in the AVR32 Architecture Manual.

PCCR - Performance Counter Control Register

Controls and configures the setup of the performance counters. The contents and functionality of the register is described in detail in the AVR32 Architecture Manual.

BEAR - Bus Error Address Register

www.BDTIC.com/ATMEL

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

SABAL - Service Access Bus Address Low

Lower part of address to Service Access Bus used by debug system.

SABAH - Service Access Bus Address High

Higher part of address to Service Access Bus used by debug system.

SABD - Service Access Bus Data

Data to or from Service Access Bus used by debug system.

www.BDTIC.com/ATMEL

2.6 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. Some of the fields in the configuration registers are fixed for all implementations using the AVR32 AP platform, while others, like the number of sets in each cache, can be different for each implementation of the platform. Such fields have IMPL in the Value field in the following tables. The programmer should refer to the data sheet for the specific product in order to obtain information on IMPL fields. The AVR32 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 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	Unused in AVR32 AP
		1	AVR32B
		Other	Reserved
AR	12:10	Architecture Revision	
		Value	Semantic
		0	Unused in AVR32 AP
		1	Revision 1
		Other	Reserved

www.BDTIC.com/ATMEL

Table 2-4. CONFIG0 Fields (Continued)

Name	Bit	Description	
MMUT	9:7	MMU type	
		Value	Semantic
		0	Unused in AVR32 AP
		1	Unused in AVR32 AP
		2	Shared TLB
		3	Unused in AVR32 AP
		Other	Reserved
F	6	Floating-point unit implemented	
		Value	Semantic
		0	No FPU implemented
		1	Unused in AVR32 AP
J	5	Java extension implemented	
		Value	Semantic
		0	Unused in AVR32 AP
		1	Java extension implemented
P	4	Performance counters implemented	
		Value	Semantic
		0	Unused in AVR32 AP
		1	Performance Counters implemented
O	3	On-Chip Debug implemented	
		Value	Semantic
		0	Unused in AVR32 AP
		1	OCD implemented
S	2	SIMD instructions implemented	
		Value	Semantic
		0	Unused in AVR32 AP
		1	SIMD instructions implemented
D	1	DSP instructions implemented	
		Value	Semantic
		0	Unused in AVR32 AP
		1	DSP instructions implemented
R	0	Memory Read-Modify-Write instructions implemented	
		Value	Semantic
		0	No RMW instructions implemented
		1	Unused in AVR32 AP

www.BDTIC.com/ATMEL

Table 2-4 shows the CONFIG1 fields.

Table 2-5. CONFIG1 Fields

<table>
<tr><th>Name</th><th>Bit</th><th colspan="2">Description</th></tr>
<tr><td>IMMU SZ</td><td>31:26</td><td colspan="2">Not used in single-MMU systems like AVR32 AP.</td></tr>
<tr><td>DMMU SZ</td><td>25:20</td><td colspan="2">Indicates the number of entries in the shared MMU in single-MMU systems like AVR32 AP. The number of entries in the MMU equals (DMMU SZ) + 1.</td></tr>
<tr><td rowspan="18">ISET</td><td rowspan="18">19:16</td><td colspan="2">Number of sets in ICACHE</td></tr>
<tr><td>Value</td><td>Semantic</td></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>2</td></tr>
<tr><td>2</td><td>4</td></tr>
<tr><td>3</td><td>8</td></tr>
<tr><td>4</td><td>16</td></tr>
<tr><td>5</td><td>32</td></tr>
<tr><td>6</td><td>64</td></tr>
<tr><td>7</td><td>128</td></tr>
<tr><td>8</td><td>256</td></tr>
<tr><td>9</td><td>512</td></tr>
<tr><td>10</td><td>1024</td></tr>
<tr><td>11</td><td>2048</td></tr>
<tr><td>12</td><td>4096</td></tr>
<tr><td>13</td><td>8192</td></tr>
<tr><td>14</td><td>16384</td></tr>
<tr><td>15</td><td>32768</td></tr>
<tr><td rowspan="10">ILSZ</td><td rowspan="10">15:13</td><td colspan="2">Line size in ICACHE</td></tr>
<tr><td>Value</td><td>Semantic</td></tr>
<tr><td>0</td><td>No ICACHE present</td></tr>
<tr><td>1</td><td>4 bytes</td></tr>
<tr><td>2</td><td>8 bytes</td></tr>
<tr><td>3</td><td>16 bytes</td></tr>
<tr><td>4</td><td>32 bytes</td></tr>
<tr><td>5</td><td>64 bytes</td></tr>
<tr><td>6</td><td>128 bytes</td></tr>
<tr><td>7</td><td>256 bytes</td></tr>
</table>

www.BDTIC.com/ATMEL

Table 2-5. CONFIG1 Fields (Continued)

Name	Bit	Description	
IASS	12:10	Associativity of ICACHE	
		Value	Semantic
		0	Direct mapped
		1	2-way
		2	4-way
		3	8-way
		4	16-way
		5	32-way
		6	64-way
		7	128-way
DSET	9:6	Number of sets in DCACHE	
		Value	Semantic
		0	1
		1	2
		2	4
		3	8
		4	16
		5	32
		6	64
		7	128
		8	256
		9	512
		10	1024
		11	2048
		12	4096
		13	8192
		14	16384
		15	32768

www.BDTIC.com/ATMEL

Table 2-5. CONFIG1 Fields (Continued)

Name	Bit	Description	
DLSZ	5:3	Line size in DCACHE	
		Value	Semantic
		0	No DCACHE present
		1	4 bytes
		2	8 bytes
		3	16 bytes
		4	32 bytes
		5	64 bytes
		6	128 bytes
		7	256 bytes
DASS	2:0	Associativity ofDCACHE	
		Value	Semantic
		0	Direct mapped
		1	2-way
		2	4-way
		3	8-way
		4	16-way
		5	32-way
		6	64-way
		7	128-way

www.BDTIC.com/ATMEL

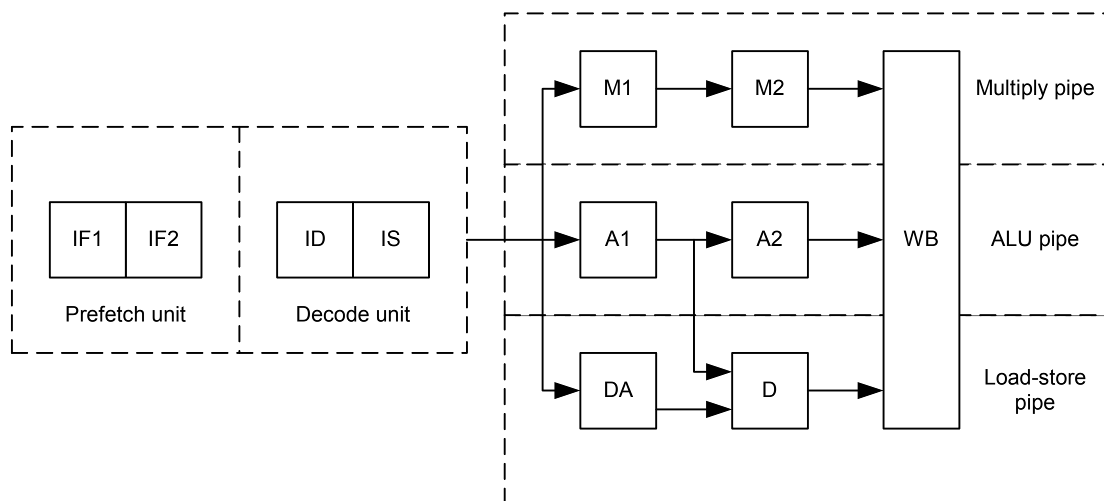
3. Pipeline

3.1 Overview

AVR32 AP is a pipelined processor with seven pipeline stages. The pipeline has three subpipes, namely the Multiply pipe, the Execute pipe and the Data pipe. These pipelines may execute different instructions in parallel. Instructions are issued in order, but may complete out of order (OOO) since the subpipes may be stalled individually, and certain operations may use a subpipe for several clock cycles.

The following figure shows an overview of the AVR32 AP pipeline stages.

Figure 3-1. The AVR32 AP pipeline stages.



The following abbreviations are used in the figure:

- IF1, IF2 - Instruction Fetch 1 and 2
- ID - Instruction Decode
- IS - Instruction Issue
- A1, A2 - ALU stage 1 and 2
- M1, M2 - Multiply stage 1 and 2
- DA - Data Address calculation stage
- D - Data cache access
- WB - Writeback

3.2 Prefetch unit

The prefetch unit comprises the IF1 and IF2 pipestages, and is responsible for feeding instructions to the decode unit. The prefetch unit fetches 32 bits at a time from the instruction cache and places them in a FIFO prefetch buffer. At the same time, one instruction, either RISC extended or compact, or Java, is fed to the decode stage.

The instruction fetches are probed for the presence of change-of-flow instructions. If such instructions are found, the prefetch unit will try to determine the destination of the instruction and continue fetching instructions from there. The branch penalty will be eliminated if the prefetch unit correctly predicts the destination of a change-of-flow instruction. When possible, the

www.BDTIC.com/ATMEL

prefetch unit will remove the change-of-flow instruction from the pipeline and replace it with the target instruction. This is called branch folding.

In Java mode, the prefetch unit is able to recognize certain Java instruction pairs and merge them together to one merged instruction. These merged instructions are passed on to ID as one instruction.

Details about the prefetch unit is given in chapter 5.

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 both for remapping register address into the different contexts, and for remapping registers to the Java operand stack if the R bit in the status register is set. The ID stage also contains the Java Operand Stack Pointer (JOSP) register which is used to address the Java operand stack if the CPU is running in Java mode.

The IS stage performs register file reads and keeps track of data hazards in the pipeline. If hazards exist, pipelines are frozen as needed in order to resolve the hazard.

3.4 ALU pipeline

The ALU pipeline performs most of the data manipulation instructions, like arithmetical and logical operations. The A1 stage performs the following tasks:

- Target address calculation and condition check for change-of-flow instructions. The A1 pipestage checks if the branch prediction performed by the prefetch unit was correct. If not, the prefetch unit is notified so that the pipeline can be flushed, the correct instruction can be fetched and the BTB can be updated.
- Condition code checking for conditional instructions.
- Address calculation for indexed memory accesses
- Writeback address calculation for the LS pipeline.
- All flag setting for arithmetical and logical instructions.
- The A2 stage performs the following tasks:
- The saturation needed by *satadd* and *satsub*.
- The operation and flag setting needed by *satrnds*, *satrndu*, *sats* and *satu*.

www.BDTIC.com/ATMEL

3.5 Multiply pipeline

All multiply instructions execute in the multiply pipeline. This pipeline contains a 32 by 16 multiplier array, and 16x16 and 32x16 multiplications therefore have an issue latency of one cycle. Multiplication of 32 by 32 bits require two iterations through the multiplier array, and therefore needs several cycles to complete. Additional cycles may be needed if an accumulation is to be performed. This will stall the multiply pipeline until the instruction is complete.

A special accumulator cache is implemented in the MUL pipeline. This cache saves the multiply-accumulate result in dedicated registers in the MUL pipeline, 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 a multiply (not MAC) instruction is executed with target address equal to that of a valid cached register, the multiply instruction will update the cache. All multiply and divide instructions will update the cache with its result, so that a subsequent MAC to the same register will not have to preload 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 updates one of the registers kept in the accumulator cache, the cache is invalidated. The cache is automatically invalidated after reset.

Some of the multiply instructions, *machh.d*, *macwh.d*, *mulwh.d* and *mulnwh.d*, produce a 48-bit result that is to be placed in two registers. These instructions all have an issue latency of 1, even though the MUL pipe only has one writeback port and two results are produced. This is handled by delaying the writeback of the low register until the MUL pipeline is idle. Then, the low register can be written back without stalling the MUL pipe. The high register is written back to the register file when the instruction leaves the M2 stage. This scheme allows several of these instructions to be issued consecutively, with no stalls due to writeback port congestion. This will increase performance in MUL-intensive applications such as DSP algorithms. The MUL pipe can only hold one delayed register for writeback, so a MUL instruction writing to another register will have to stall one cycle in IS if a writeback is pending in the MUL pipe. Hazard detection is performed on the pending writeback register, so any instruction reading a register pending writeback will stall in IS until the value is forwardable in M2.

The multiply pipeline also contains a divider, performing multicycle 32-by-32 signed and unsigned division with both quotient and remainder outputs.

In general, the MUL instructions do not set any flags. However, some of the MUL instructions may set the saturate (Q) flag. No hazard detection is performed on this setting of the Q flag. **The programmer must ensure that such a Q flag update has propagated to the status register before using the Q flag.**

www.BDTIC.com/ATMEL

3.6 Load-store pipeline

The load-store (LS) pipeline is able to read or write up to two registers per clock cycle, if the data is 64-bit aligned. The address is calculated by the A1 pipe stage for indexed and load-extracted-index accesses, the DA stage performs all other address calculations. Thereafter the address is passed on to the LS pipe and output to the cache, 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 cache in the D stage. If the read data requires typecasting or other manipulation like performed by *ldins* or *ldswp*, this manipulation is performed in the WB stage.

The LS pipeline also contains hardware for performing load and store multiple instructions decoupled from the rest of the core. For such instructions, the A1 stage calculates the pointer writeback address if needed. The load or store is then decoupled from the integer unit, and the integer unit may execute sequential instructions if no hazards occur. Load and store of multiple registers are performed by accessing 2 words at a time. If the first address is not 64-bit aligned, the first access is performed as a single word. The rest of the transfer is then performed as 64 bit accesses. The last transfer may need to be performed as a 32 bit access, depending on the number of registers to load or store.

For code efficiency purposes, the programmer should always try to rearrange the instructions in the code in such a way that no data stalls will occur.

3.6.1 Support for unaligned addresses

The LS pipeline is able to perform certain word-sized load and store instructions of any alignment, and word-aligned *st.d* and *ld.d*. Any other unaligned memory access will cause an MMU address exception. All coprocessor memory access instructions require word-aligned pointers. Doubleword-sized accesses with word-aligned pointers will automatically be performed as two word-sized accesses.

The following table shows the instructions with support for unaligned addresses. All other instructions require aligned addresses. Accessing an unaligned address may require several clock cycles, refer to Section 10. on page 154 for details.

Table 3-1. Instructions with unalignment support

Instruction	Supported alignment
ld.w	Any
st.w	Any
lddsp	Any
lddpc	Any
stdsp	Any
ld.d	Word
st.d	Word
All coprocessor memory access instruction	Word

www.BDTIC.com/ATMEL

3.7 Writeback

The three subpipes share a writeback (WB) stage with three register file write ports. If the three subpipes produces four results at the same time, the MUL pipeline is temporarily stalled until a writeback port is available. The WB stage also contains logic for:

- Sign- or zero-extention of data loaded from cache.
- Execution of *ldins* and *ldswp*.
- Output formatting of data loaded from unaligned addresses.

3.8 Forwarding hardware and hazard detection

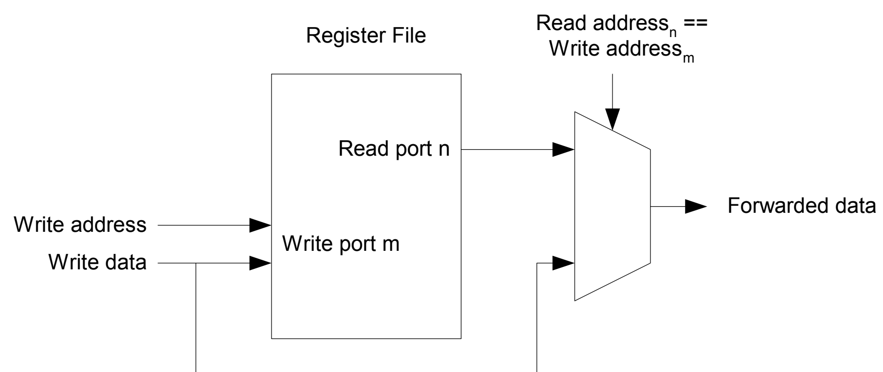
The pipeline is implemented in such a way that the programmer in most cases will not have to consider hazards between instructions when writing code. Efficient operand forwarding mechanisms are implemented in order to minimize pipeline stalls due to data dependencies. When dependencies exist, the hardware will stall the affected parts of the pipeline in order to guarantee correct execution. Data forwarding is done automatically and is invisible to the user. This ensures that all code will execute correctly, even though the pipeline may have to be stalled in some cases. The user should be aware of these stalls and try to rewrite the code so that no such dependencies arise. This will result in faster execution.

Since instructions are allowed to complete out of order, both Write-After-Read (WAR), Write-After-Write (WAW) and Read-After-Write (RAW) hazards may occur. If an instruction is affected by a hazard, or will provoke a hazard, it is frozen in the IS stage until the hazard is resolved. This will also freeze all upstream pipeline stages. All downstream stages are allowed to continue execution. Instructions storing data to memory will read the data to store from the register file in the D pipeline stage. This pipeline stage has a dedicated hazard detection and forwarding unit. If the data to store to memory is not available in the D stage, the LS pipe will have to stall. Newer instructions may still start executing in the other pipelines.

3.8.1 IS stage forwarding

The IS stage is able to forward data from the register file inputs to the register file outputs. If data to write is present at the write ports of the register file at the same time as the register is read, the data not yet written will be read. This ensures that data from the writeback stages are forwarded to the register file outputs. This is illustrated in Figure 3-2:

Figure 3-2. Forwarding inside the IS stage

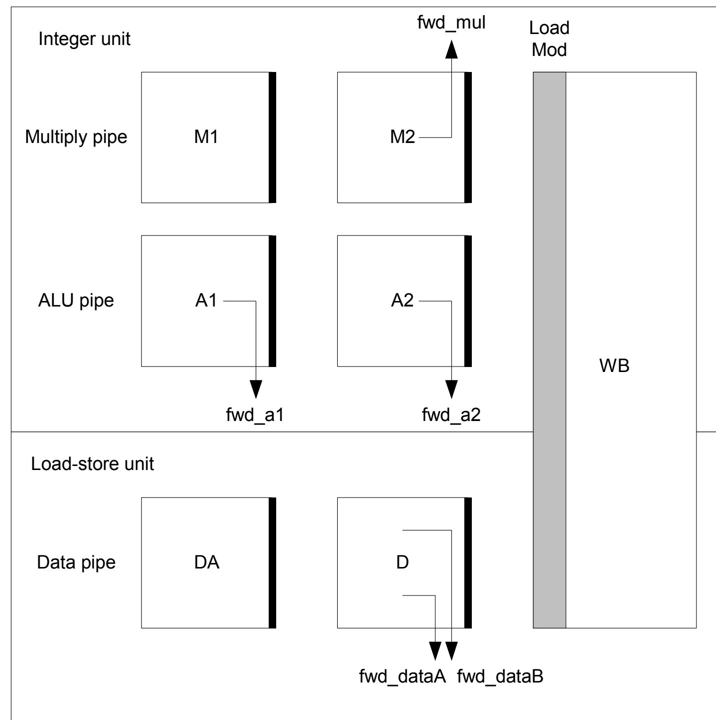


www.BDTIC.com/ATMEL

3.8.2 Forwarding sources

All operations that produce valid results are forwarded. All data are forwarded directly from the inputs of pipeline registers. The following figure shows the forwarding sources, and the name of the forwarded signals. Each of the forwarded signals carry a word-sized value. Pipeline registers are illustrated as a thick black line, the load modification unit is illustrated as a gray box.

Figure 3-3. Forwarding sources

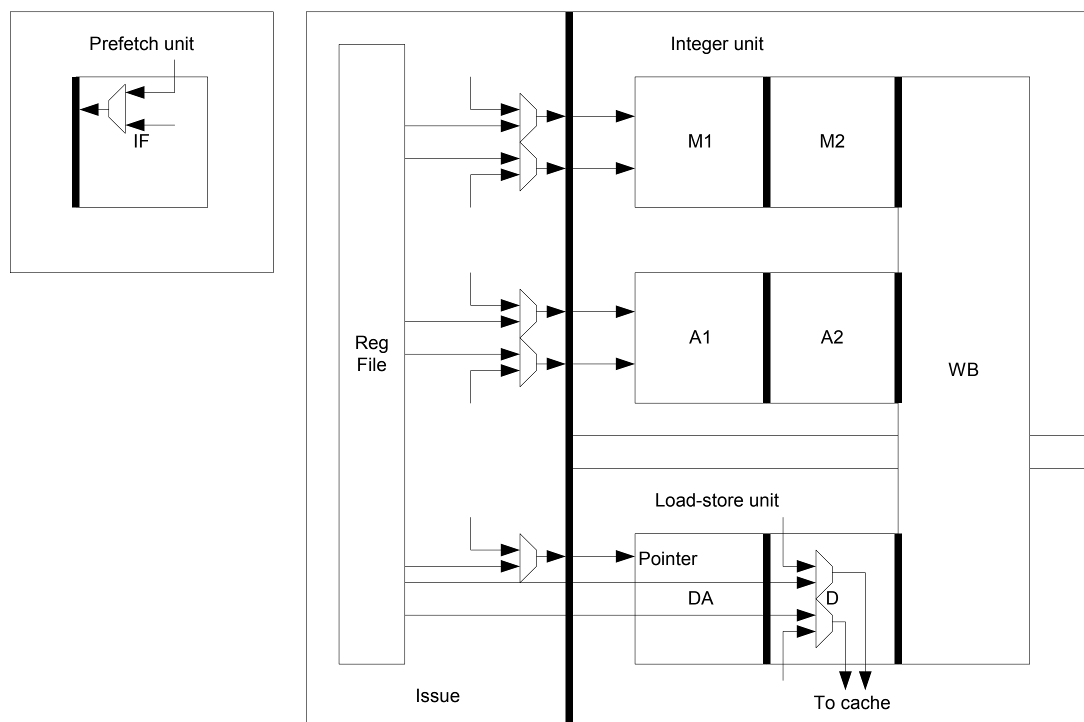


3.8.3 Forwarding destinations

The forwarded data is input to the IS stage. The IS stage has logic deciding whether the value read from the register file is valid, or if a forwarded value should be used. This is illustrated in Figure 3-4. Forwarded data is shown with bent arrows, and data from the previous pipeline stage is shown in straight arrows. The forwarded value really consists of all the possible forward values described in Figure 3-3, but is shown as a single value for simplicity. The prefetch unit also receives forwarded data. This data is used for calculating an instruction fetch address for change-of-flow instructions. Target addresses for change-of-flow instructions are produced either by the A1 stage, or the WB stage.

www.BDTIC.com/ATMEL

Figure 3-4. Forwarding destinations



3.9 Hazards not handled by the hardware

All hazards occurring between normal arithmetical, logical, load-store and change-of-flow instructions are handled automatically by hardware. There are, however, a few instruction sequences which must be sequenced by the user. These sequences are described in this chapter. The programmer can assume that any instruction sequence other than the sequences explicitly mentioned in this chapter will work without any special consideration.

3.9.1 Accessing system registers with mtsr and mfsr

The *mtsr* instruction writes the contents of a register into a system register. The system registers control the behaviour of the CPU. The programmer must make sure that any *mtsr* instruction has committed and has altered the state of the system in the desired way before issuing any new instructions that depend on this new state. This can be done by inserting *nop* instructions, or other instructions that do not depend on the new state generated by the *mtsr* instruction.

Table 2-2, "System Registers implemented in AVR32 AP," on page 10 details the timing for writes into the different system registers. The system registers are written as the *mtsr* instruction leaves the pipeline stage described in the table. The system registers are read as the *mfsr* instruction leaves the pipeline stage described in the table. As soon as a system register is read by *mfsr*, it can be forwarded as any regular register file register.

Some of the system registers are located inside modules on the TCB bus. These are written when the *mtsr* instruction leaves the D pipeline stage. Instructions depending on a *mtsr* to these system registers being committed must therefore wait in the IS stage until the effects of the *mtsr* is guaranteed to be visible to the instruction. The following code demonstrates a write to the ASID field of TLBEHI, followed by a *rete* to an address which requires the new ASID to be visible. A *nop* is inserted to guarantee that the *mtsr* leaves the D stage at the same time as *rete* leaves the A1 stage. In the following cycle, the icache will start fetching at the specified address

www.BDTIC.com/ATMEL

and observe the newly updated ASID. Register r0 is assumed to contain the correct value to write into TLBEHI.

```
mtsr TLBEHI, r0
nop
rete
```

3.9.2 Writing to the status register with ssrf and csrf

These instructions have the same timing as a *mtsr* to the system register.

3.9.3 Writing to and using the JOSP register

The JOSP register is used to determine which register file register to access when in Java mode. This is needed because the 8 elements on top of the Java operand stack are located in the register file. Since the register addresses are generated in the ID stage, JOSP is located here.

JOSP is automatically updated to the correct value when executing Java bytecodes in Java mode. One may also need to update the JOSP register manually, either with the *incjosp* instruction, or using *mtsr*/*mfsr* for reading/writing JOSP.

When updating JOSP with *incjosp*, JOSP is updated with the new value when *incjosp* has left ID. The *incjosp* instruction reads the value of JOSP when it is in ID, and writes the new value as it leaves ID. If the *incjosp* instruction is flushed from the pipe before being committed for some reason like an interrupt or a taken change-of-flow instruction, hardware automatically restores the correct value to the JOSP register. The JOSP register will be restored to the value it had after the last completed instruction.

When updating JOSP with *mtsr*, JOSP is updated with the new value when *mtsr* has left A1. The user is responsible for not letting any instruction that uses JOSP leave ID before mtsr has written the new JOSP value. This may require inserting *nop* instructions between *mtsr* and any instruction using JOSP.

The following assembly code illustrates coding to avoid hazards when accessing JOSP. Two *nop* instructions are inserted to make sure that the new value of JOSP written by *mtsr* as *mtsr* leaves A1 is visible to the *incjosp* instruction when it enters ID. A *mfsr* instruction may follow immediately after *incjosp*, as *incjosp* writes the new JOSP value when it leaves ID, while *mfsr* reads JOSP while it is in A1.

```
mtsr JOSP, r0
nop
nop
incjosp -2
mfsr r1, JOSP
```

The following assembly code is another illustration of coding to avoid hazards when accessing JOSP. The two sets of code perform identical operations. This code sets the R bit in the status register in order to enable remapping of the register file to a Java operand stack. This effectively remaps r0 to r7 into a Java operand stack, where the mapping from logical register to physical register is dependent on the value of JOSP. Note that the second code example is strongly discouraged to use in practice, since no JOSP over/underflow detection is performed. The code is presented only to show the differences in timing between the two ways of writing to JOSP.

In the first code, *incjosp* changes the value of JOSP when it is in ID. The new value of JOSP is therefore visible when the *add* instruction enters ID.

www.BDTIC.com/ATMEL

In the second code, *mtsr* writes the new value of JOSP as it leaves A1. As the *add* instruction needs JOSP to be updated when it enters ID because of the register remapping, two *nop* instructions must be inserted.

```
ssrf R
incjosp -2
add  r0, r0


ssrf R
mfsr r8, JOSP
sub  r8, 2
mtsr JOSP, r8
nop
nop
add  r0, r0
```

3.9.4 Execution of TLB instructions

The TLB instructions *tlbr*, *tlbw* and *tlbs* are used to maintain the data in the TLB. They use the TCB bus to access the MMU, and the instruction is dispatched to the MMU when the instruction is in the D pipeline stage. The programmer must make sure that any writes to the TLB with the *tlbw* instructions are completed before the TLB entry is used in an icache or dcache memory access. This is handled automatically for any dcache memory access, since any load/store instructions flow through the same pipeline as the *tlbw* instruction, and the *tlbw* instruction will have left the D stage before any load/store instruction enters it. Any icache access that is to use the page table entry written by *tlbw* must wait until the *tlbw* instruction is in the D pipeline stage. This may require inserting a *nop* or another unrelated instruction, as illustrated in the code below, which shows a part of a ITLB miss handler. The *rete* instruction wishes to use the page table entry written by *tlbw* to generate the physical address of the instruction to return to.

```
tlbw
nop
rete
```

3.9.5 Execution of cache instructions

The *cache* instruction perform various cache-relatated operations, like invalidation of lines. Some of these operations are harmless, and need no sequencing or hazard consideration. Other operations, like invalidation, require more concern. The programmer must make sure that any invalidation is committed before any instruction the depends on the invalidation already being performed is allowed to execute.

The *cache* instruction use the TCB bus to access the caches, and the instruction is dispatched to the caches when the instruction is in the D pipeline stage. The programmer must make sure that any *cache* instructions are completed before any icache or dcache memory access that depends on the cache instruction is executed. This is handled automatically for any dcache memory access, since any load/store instructions flow through the same pipeline as the *cache* instruction, and the *cache* instruction will have left the D stage before any load/store instruction enters it. Any icache access that is dependent on the *cache* instruction must wait until the *cache* instruction is in the D pipeline stage. This may require inserting a *nop* or another unrelated instruction, as illustrated in the code below. The *rjmp* instruction wishes to jump to a location

ATMEL®
www.BDTIC.com/ATMEL

labeled flushedaddress that must be flushed from the cache. INVALIDATEI is a macro that is defined to be the command for invalidation of the icache.

```
cache  INVALIDATEI
nop
rjmp flushedaddress
```

3.9.6 Hazards on the Q flag

Some of the instructions in the instruction set updates the status register Q flag. Many of these instructions, like satadd, generate the new Q flag after a single cycle so no hazards are present between these instructions and other instructions. The *sats*, *satu*, *satrnds*, *satrndu* and some multiply instructions, require several cycles before updating the Q flag. The required Q flag latency for each of these instructions is listed in Section 10. on page 154. The user must make sure that any of these instructions have completed and updated the Q flag before using the Q flag in any computations. In the following example, a *satrnds* instruction is followed by a branch-if-q-set instruction. A *nop* is needed in order to guarantee correct execution.

```
satrnds r0>>0, 5
nop
brqs targetaddress
```

3.10 Event handling

The CPU is able to respond to different events. An event can be either an interrupt or an exception. Interrupts are requests from external modules and are routed through the interrupt controller. Exceptions are system events that require handling outside normal program flow.

Different types of exceptions can occur during execution of an instruction. Some exceptions are instruction-address related, and occur during instruction fetch. Other exceptions occur during decode, like unimplemented instruction and illegal opcode. Data access instructions can cause data-address related exceptions, like DTLB miss. Exceptions can occur in different pipe stages, depending on the type of exception. Several exceptions can be related to the same instruction. Mechanisms must therefore be implemented so that several exceptions associated with the same instruction can be handled correctly. The exception priorities are defined Table 3-2 on page 34. An instruction that has caused an exception request is called a contaminated instruction.

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.

Events are detected in two different pipeline stages. The D stage detects all data-address related exceptions (DTLB multiple hit, DTLB miss, DTLB protection and DTLB modified). All other exceptions and interrupts are detected in the A1 stage. Data breakpoints are also detected in A1.

A complication occurs with the event detection in the A1 stage: The instruction tagged as contaminated may be part of a folded branch. In this case, the event is taken only if the branch prediction was correct. Otherwise, the entire folded branch instruction is flushed.

Data-address related exceptions are detected in the D stage. The address boundary check unit ensures that no sequential instructions are issued unless it can be guaranteed that the data access will not generate an exception.

www.BDTIC.com/ATMEL

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. When an event is accepted, the A1 stage and all upstream stages are flushed.

Branch folding complicates exception handling. If a folded instruction fails the condition check in the A1 stage, the address of the folded instruction should be used as restart address. This is implemented by passing the address of the folded instruction in the PC pipeline register.

When folding branches, both the branch and the folded instruction can be contaminated. How do we determine which of the two instructions caused the exception? The fetch stage is responsible for not folding instructions if the branch instruction is contaminated. The branch instruction can be contaminated only due to instruction-address related exceptions, as it must already have been decoded and recognized in order to have been placed in the BTB. This contamination is known already in IF. If folding has occurred, it is guaranteed that the contamination was not in the branch instruction, and it must therefore be in the folded instruction. Therefore, the folded instruction should be restarted.

3.10.1 Event priority

Several instructions may be in the pipeline at the same time, and several events may be issued in each pipeline stage. This implies that several pending exceptions may be in the pipeline simultaneously. Priorities must therefore be imposed, ensuring that the correct event is serviced first. The priority scheme obeys the following rules:

1. If several instructions trigger events, the instruction furthest down the pipeline is serviced first, even if upstream instructions have pending events of higher priority.
2. If this instruction has several pending events, the event with the highest priority is serviced first. After this event has been serviced, all pending events are cleared and the instruction is restarted.

3.10.2 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 responsability 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 in the Return Status Register and Return Address Register corresponding to the new context. Saving 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.

www.BDTIC.com/ATMEL

3. The Mode bits are set correctly to reflect the priority of the accepted event, and the correct register file banks are selected. The address of the event handler, as shown in Table 3-2, is loaded into the Program Counter.

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

The *rete* instruction signals the end of the event. When encountered, the values in the Return Status Register and Return Address Register corresponding to the event context are restored to the Status Register and Program Counter. The restored Status Register contains information allowing the core to resume operation in the previous execution mode. This concludes the event handling.

3.10.3 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 the Architecture Manual. In order to allow the *scall* routine to return to the correct context, a return from supervisor call instruction, *rets*, is implemented.

3.10.4 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. The mode bits in the status register can freely be manipulated in Debug mode, to observe registers in all contexts, while retaining full privileges.

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

3.11 Entry points for events

Several different event handler entry points exists. For AVR32B, the reset routine entry address is always fixed to 0xA000_0000. This address resides in unmapped, uncached space in order to ensure well-defined resets.

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.

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 the privileged address space, or in a

www.BDTIC.com/ATMEL

privileged memory protection region. In a segmented AVR32B system, some segments of the virtual memory space may be better suited than others for holding event handlers. This is due to differences in translateability and cacheability between segments. A cacheable, non-translated segment may offer the best performance for event handlers, as this will eliminate any TLB misses and speed up instruction fetch. The user may also consider to lock the event handlers in the instruction cache.

If several events occur on the same instruction, they are handled in a prioritized way. The priority ordering is presented in Table 3-2. 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 3-2 on page 34

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). In AVR32 AP, the actual autovector address is formed by bitwise OR-ing the autovector offset to EVBA. Using bitwise-OR instead of an adder saves hardware. The programmer must consider this when setting up EVBA.

www.BDTIC.com/ATMEL

Table 3-2. Priority and handler addresses for events

Priority	Handler Address	Name	Event source	Stored Return Address
1	0xA000_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	Internal signal	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	ITLB	PC of offending instruction
13	EVBA+0x50	ITLB Miss	ITLB	PC of offending instruction
14	EVBA+0x18	ITLB Protection	ITLB	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 in AVR32 AP
20	EVBA+0x30	Coprocessor absent	Instruction	PC of offending instruction
21	EVBA+0x100	Supervisor call	Instruction	PC(Supervisor Call) +2
22	EVBA+0x34	Data Address (Read)	DTLB	PC of offending instruction
23	EVBA+0x38	Data Address (Write)	DTLB	PC of offending instruction
24	EVBA+0x60	DTLB Miss (Read)	DTLB	PC of offending instruction
25	EVBA+0x70	DTLB Miss (Write)	DTLB	PC of offending instruction
26	EVBA+0x3C	DTLB Protection (Read)	DTLB	PC of offending instruction
27	EVBA+0x40	DTLB Protection (Write)	DTLB	PC of offending instruction
28	EVBA+0x44	DTLB Modified	DTLB	PC of offending instruction

www.BDTIC.com/ATMEL

3.11.1 Description of events in AVR32 AP

3.11.1.1 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 0xA000_0000.

```
SR = reset_value_of_SREG;
PC = 0xA000_0000;
```

All other system registers are reset to their reset value, which may or may not be defined. Refer to "Programming Model" on page 6 for details.

3.11.1.2 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[R] = 0;
SR[J] = 0;
SR[D] = 1;
SR[DM] = 1;
SR[EM] = 1;
SR[GM] = 1;
```

3.11.1.3 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.

```
RSR_EX = SR;
RAR_EX = PC of offending instruction;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x00;
```


www.BDTIC.com/ATMEL

3.11.1.4 TLB Multiple Hit Exception

TLB Multiple Hit exception is issued when multiple address matches occurs in the TLB, causing an internal inconsistency.

This exception signals a critical error where the hardware is in an undefined state. All interrupts are masked, and PC is loaded with EVBA | 0x04. MMU-related registers are updated with information in order to identify the failing address and the failing TLB if multiple TLBs are present. TLBEHI[ASID] is unchanged after the exception, and therefore identifies the ASID that caused the exception.

```
RSR_EX = SR;
RAR_EX = PC of offending instruction;
TLBEAR = FAILING_VIRTUAL_ADDRESS;
TLBEHI[VPN] = FAILING_PAGE_NUMBER;
TLBEHI[I] = 0/1, depending on which TLB caused the error;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x04;
```

3.11.1.5 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 value placed in RAR_EX is unrelated to the operation that caused the exception. The exception handler is responsible for performing the appropriate action.

```
RSR_EX = SR;
RAR_EX = PC of first non-issued instruction;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x08;
```

3.11.1.6 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.

```
RSR_EX = SR;
RAR_EX = PC;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
```


www.BDTIC.com/ATMEL

```
SR[GM] = 1;
PC = EVBA | 0x0C;
```

3.11.1.7 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.

```
RSR_NMI = SR;
RAR_NMI = Address of first noncompleted instruction;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'111;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x10;
```

3.11.1.8 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.

```
RSR_INT3 = SR;
RAR_INT3 = Address of first noncompleted instruction;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'101;
SR[I3M] = 1;
SR[I2M] = 1;
SR[I1M] = 1;
SR[I0M] = 1;
PC = EVBA | INTERRUPT_VECTOR_OFFSET;
```

3.11.1.9 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

www.BDTIC.com/ATMEL

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.

```
RSR_INT2 = SR;
RAR_INT2 = Address of first noncompleted instruction;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'100;
SR[I2M] = 1;
SR[I1M] = 1;
SR[I0M] = 1;
PC = EVBA | INTERRUPT_VECTOR_OFFSET;
```

3.11.1.10 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.

```
RSR_INT1 = SR;
RAR_INT1 = Address of first noncompleted instruction;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'011;
SR[I1M] = 1;
SR[I0M] = 1;
PC = EVBA | INTERRUPT_VECTOR_OFFSET;
```

3.11.1.11 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.

www.BDTIC.com/ATMEL

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.

```
RSR_INT0 = SR;
RAR_INT0 = Address of first noncompleted instruction;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'010;
SR[I0M] = 1;
PC = EVBA | INTERRUPT_VECTOR_OFFSET;
```

3.11.1.12 Instruction Address Exception

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

```
RSR_EX = SR;
RAR_EX = PC;
TLBEAR = FAILING_VIRTUAL_ADDRESS;
TLBEHI[VPN] = FAILING_PAGE_NUMBER;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x14;
```

3.11.1.13 ITLB Miss Exception

The ITLB Miss exception is generated when no TLB entry matches the instruction memory address, or if the Valid bit in a matching entry is 0.

```
RSR_EX = SR;
RAR_EX = PC;
TLBEAR = FAILING_VIRTUAL_ADDRESS;
TLBEHI[VPN] = FAILING_PAGE_NUMBER;
TLBEHI[I] = 1;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x50;
```


www.BDTIC.com/ATMEL

3.11.1.14 *ITLB Protection Exception*

The ITLB Protection exception is generated when the instruction memory access violates the access rights specified by the protection bits of the addressed virtual page.

```
RSR_EX = SR;
RAR_EX = PC;
TLBEAR = FAILING_VIRTUAL_ADDRESS;
TLBEHI[VPN] = FAILING_PAGE_NUMBER;
TLBEHI[I] = 1;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x18;
```

3.11.1.15 *Breakpoint Exception*

The Breakpoint exception is issued when a *breakpoint* instruction is executed, or the OCD breakpoint input line to the CPU is asserted, and SREG[DM] is cleared.

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[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[D] = 1;
SR[DM] = 1;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x1C;
```

3.11.1.16 *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.

```
RSR_EX = SR;
RAR_EX = PC;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x20;
```


www.BDTIC.com/ATMEL

3.11.1.17 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.

```
RSR_EX = SR;
RAR_EX = PC;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x24;
```

3.11.1.18 Data Read Address Exception

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

```
RSR_EX = SR;
RAR_EX = PC;
TLBEAR = FAILING_VIRTUAL_ADDRESS;
TLBEHI[VPN] = FAILING_PAGE_NUMBER;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x34;
```

3.11.1.19 Data Write Address Exception

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

```
RSR_EX = SR;
RAR_EX = PC;
TLBEAR = FAILING_VIRTUAL_ADDRESS;
TLBEHI[VPN] = FAILING_PAGE_NUMBER;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x38;
```


www.BDTIC.com/ATMEL

3.11.1.20 *DTLB Read Miss Exception*

The DTLB Read Miss exception is generated when no TLB entry matches the data memory address of the current read operation, or if the Valid bit in a matching entry is 0.

```
RSR_EX = SR;
RAR_EX = PC;
TLBEAR = FAILING_VIRTUAL_ADDRESS;
TLBEHI[VPN] = FAILING_PAGE_NUMBER;
TLBEHI[I] = 0;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x60;
```

3.11.1.21 *DTLB Write Miss Exception*

The DTLB Write Miss exception is generated when no TLB entry matches the data memory address of the current write operation, or if the Valid bit in a matching entry is 0.

```
RSR_EX = SR;
RAR_EX = PC;
TLBEAR = FAILING_VIRTUAL_ADDRESS;
TLBEHI[VPN] = FAILING_PAGE_NUMBER;
TLBEHI[I] = 0;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x70;
```

3.11.1.22 *DTLB Read Protection Exception*

The DTLB Protection exception is generated when the data memory read violates the access rights specified by the protection bits of the addressed virtual page.

```
RSR_EX = SR;
RAR_EX = PC;
TLBEAR = FAILING_VIRTUAL_ADDRESS;
TLBEHI[VPN] = FAILING_PAGE_NUMBER;
TLBEHI[I] = 0;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x3C;
```


www.BDTIC.com/ATMEL

3.11.1.23 DTLB Write Protection Exception

The DTLB Protection exception is generated when the data memory write violates the access rights specified by the protection bits of the addressed virtual page.

```
RSR_EX = SR;
RAR_EX = PC;
TLBEAR = FAILING_VIRTUAL_ADDRESS;
TLBEHI[VPN] = FAILING_PAGE_NUMBER;
TLBEHI[I] = 0;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x40;
```

3.11.1.24 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 3-3. When entering the exception routine, the address of the instruction that caused the exception is stored as yhe stacked return address.

```
RSR_EX = SR;
RAR_EX = PC;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x28;
```


www.BDTIC.com/ATMEL

Table 3-3. 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
cache - perform cache operation	
tlbr - read addressed TLB entry into TLBEHI and TLBELO	
tlbw - write TLB entry registers into TLB	
tlbs - search TLB for entry matching TLBEHI[VPN]	
mtsr - move to system register	Unpriviledged when accessing JOSP and JECR
mfsr - move from system register	Unpriviledged when accessing JOSP and JECR
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

3.11.1.25 DTLB Modified Exception

The DTLB Modified exception is generated when a data memory write hits a valid TLB entry, but the Dirty bit of the entry is 0. This indicates that the page is not writable.

```
RSR_EX = SR;
RAR_EX = PC;
TLBEAR = FAILING_VIRTUAL_ADDRESS;
TLBEHI[VPN] = FAILING_PAGE_NUMBER;
TLBEHI[I] = 0;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x44;
```


www.BDTIC.com/ATMEL

3.11.1.26 Floating-point Exception

The Floating-point exception is generated when the optional Floating-Point Hardware signals that an IEEE® exception occurred, or when another type of error from the floating-point hardware occurred. Unused in AVR32 AP since it has no FP hardware.

```
RSR_EX = SR;
RAR_EX = PC;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x2C;
```

3.11.1.27 Coprocessor Exception

The Coprocessor exception occurs when the addressed coprocessor does not acknowledge an instruction. This permits software implementation of coprocessors.

```
RSR_EX = SR;
RAR_EX = PC;
SR[R] = 0;
SR[J] = 0;
SR[M2:M0] = B'110;
SR[EM] = 1;
SR[GM] = 1;
PC = EVBA | 0x30;
```

3.11.1.28 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} )
  RAR_SUP ← PC + 2;
  RSR_SUP ←  SR;
  PC ← EVBA | 0x100;
  SR[M2:M0] ← B'001;
else
  LR_Current Context ← PC + 2;
  PC ← EVBA | 0x100;
```


www.BDTIC.com/ATMEL

3.12 Interrupt latencies

The following features in AVR32 AP ensure low and deterministic interrupt latency:

- Four different interrupt levels and an NMI ensures that the user can efficiently prioritize the interrupt sources.
- Interrupts are autovectored, allowing the CPU to jump directly to the interrupt handler.
- A shadowed interrupt context for INT3 is provided so that critical interrupt handlers can start directly without having to stack registers.
- Interrupt handler code can be locked in the icache, and the corresponding page table information can be locked in the TLB.

The following calculations makes the following assumptions:

- The interrupt handler code is present in the icache and fetching handler instructions does not cause any MMU exceptions.
- The pending interrupt is of higher priority than any executing interrupts, so that it can be handled immediately.
- Any instructions in DA or D do not cause a cache miss. Any interrupts will wait until instructions in DA or D have left these pipeline stages. If the instruction in DA or D cause a cache miss, the time for the cache line to be loaded so that the instruction can complete will depend on the timing of the memory the data will be loaded from. Any time spent reloading a cache line must be added to the maximum interrupt latency calculated below.

3.12.1 Maximum interrupt latency

The maximum interrupt latency occurs when a long-running instruction is present in DA. Any instruction must have left DA and D before interrupt handling will commence. The latency can be calculated as follows:

Table 3-4. Maximum interrupt latency

Source	Delay
Wait for the slowest instruction (ldm/stm) to leave DA and D	10
Wait for autovector target instruction to be fetched	4
TOTAL	14

3.12.2 Minimum interrupt latency

The maximum interrupt latency can be calculated as follows:

Table 3-5. Maximum interrupt latency

Source	Delay
DA and D are empty	0
Wait for autovector target instruction to be fetched	4
TOTAL	4

www.BDTIC.com/ATMEL

3.13 Processor consistency

Special hardware is implemented ensuring strict processor consistency, despite the use of OOO completion. No instruction is allowed to change the state of the processor if there is a possibility that an older, uncommitted instruction may not complete. In such a case, the younger instruction is frozen in the IS stage until it can be guaranteed that the older instruction will commit. In practice, it is only memory access instructions that can cause a recoverable exception after they have left the IS stage. Such address-related exceptions are always detected at the end of the D stage. All other exceptions occuring after an instruction has left the IS stage are unrecoverable, so processor consistency is unimportant, as a reset will have to be performed anyway.

The following mechanisms ensure processor consistency:

3.13.1 Address boundary checking

If a memory access instruction generates addresses that cross a page boundary, the next sequential instruction is frozen in the IS stage until the memory access instruction has successfully left the D stage. This ensures that no address-related exceptions will occur in the middle of a memory access instruction. As a consequence, the memory access instruction is guaranteed to complete, and the following instruction may safely leave the IS stage.

Simple address checking is used to ensure that a memory access instruction cannot cause an address related exception. This is checked by examining the memory pointer, the size of the data transfer and the direction of pointer incrementation.

3.13.2 Handling contaminated instructions

Contaminated instructions are instructions that are tagged as having caused an exception. The following rules ensures in-order completion and handling of contaminated instructions.

- A contaminated instruction is frozen in the IS stage until the DA and D stages are empty.
- When a contaminated instruction leaves the IS stage, it is issued to the A1 stage, regardless of instruction type. All sequential instructions are frozen until the contaminated instruction has either committed or been flushed from the pipe. This last event can occur only when the contaminated instruction is folded with a branch.

3.13.3 Handling instructions with PC as destination

Instructions with PC as destination register will cause a change of flow. It must therefore be ensured that no sequential instructions are allowed to commit before the instruction updating the PC. When the instruction updating PC has left IS, all upstream stages are frozen until the instruction updating PC has committed. The new PC value is forwarded directly from the WB stage to the IF stage.

www.BDTIC.com/ATMEL

4. Virtual memory

The AVR32 architecture uses virtual memory in order to support operating systems and large memory spaces efficiently. Virtual memory simplifies execution of multiple processes and allows allocation of privileges to different sections of the memory space.

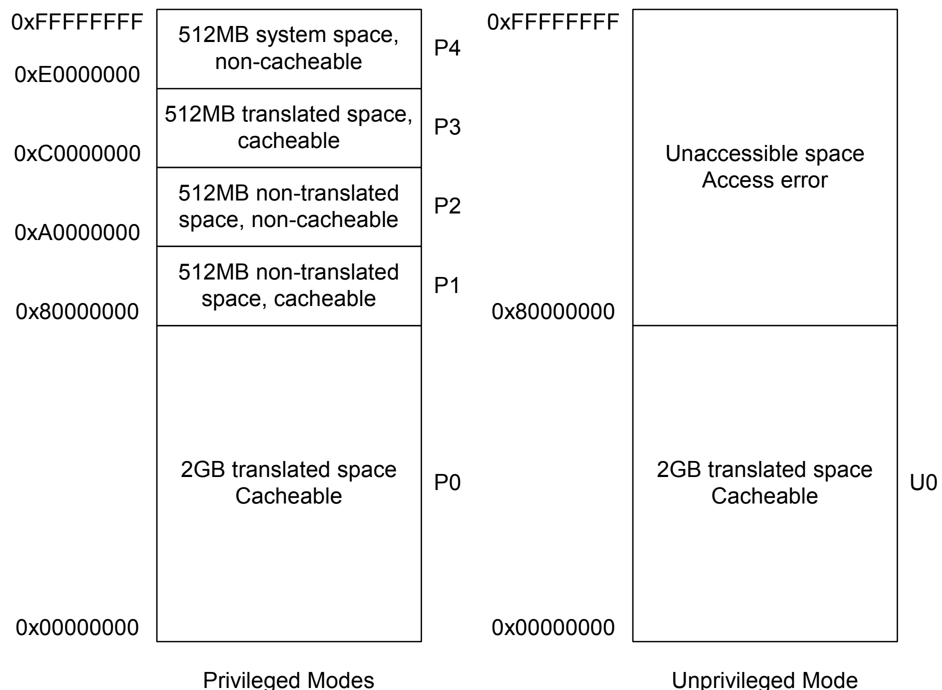
The AVR32 architecture specifies a 32-bit virtual memory space. This virtual space can be mapped to a 32-bit physical space. How this memory space is used and mapped is defined by bus controllers and memory controllers on the outside of AVR32 AP.

4.1 Memory map

The memory map has six different segments, named P0 through P4, and U0. The P-segments are accessible in the privileged modes, while the U-segment is accessible in the unprivileged mode.

The virtual memory map is specified below.

Figure 4-1. The AVR32 virtual memory space



Both the P1 and P2 segments are default segment translated to the physical address range 0x00000000 to 0x1FFFFFFF. The mapping between virtual addresses and physical addresses is therefore implemented by clearing of MSBs in the virtual address. The difference between P1 and P2 is that P1 is cached, while P2 is uncached. Because P1 and P2 are segment translated and not page translated, code for initialization of MMUs and exception vectors are located in these segments. P1, being cacheable, offers higher performance than P2.

The P3 space is also by default segment translated to the physical address range 0x00000000 to 0x1FFFFFFF. By enabling and setting up the MMU, the P3 space becomes page translated. Page translation will override segment translation.

The P4 space is intended for memory mapping special system resources like peripheral modules. This segment is non-cacheable, non-translated.

www.BDTIC.com/ATMEL

The U0 segment is accessible in the unprivileged user mode. This segment is cacheable and translated, depending upon the configuration of the cache and the memory management unit. If accesses to other memory addresses than the ones within U0 is made in application mode, an access error exception is issued.

The virtual address map is summarized in Table 4-1.

Table 4-1. The virtual address map

Virtual address [31:29]	Segment name	Virtual Address Range	Segment size	Accessible from	Default segment translated	Characteristics
111	P4	0xFFFF_FFFF to 0xE000_0000	512 Mb	Privileged	No	System space Unmapped, Uncacheable
110	P3	0xDFFF_FFFF to 0xC000_0000	512 Mb	Privileged	Yes	Mapped, Cacheable
101	P2	0xBFFF_FFFF to 0xA000_0000	512 Mb	Privileged	Yes	Unmapped, Uncacheable
100	P1	0x9FFF_FFFF to 0x8000_0000	512 Mb	Privileged	Yes	Unmapped, Cacheable
0xx	P0 / U0	0x7FFF_FFFF to 0x0000_0000	2 Gb	Unprivileged Privileged	No	Mapped, Cacheable

The segment translation can be disabled by clearing the S bit in the MMUCR. This will place all the virtual memory space into a single 4 GB mapped memory space. Segment translation is enabled by default.

The AVR32 architecture has two translations of addresses.

1. Segment translation (enabled by the MMUCR[S] bit)
2. Page translation (enabled by the MMUCR[E] bit)

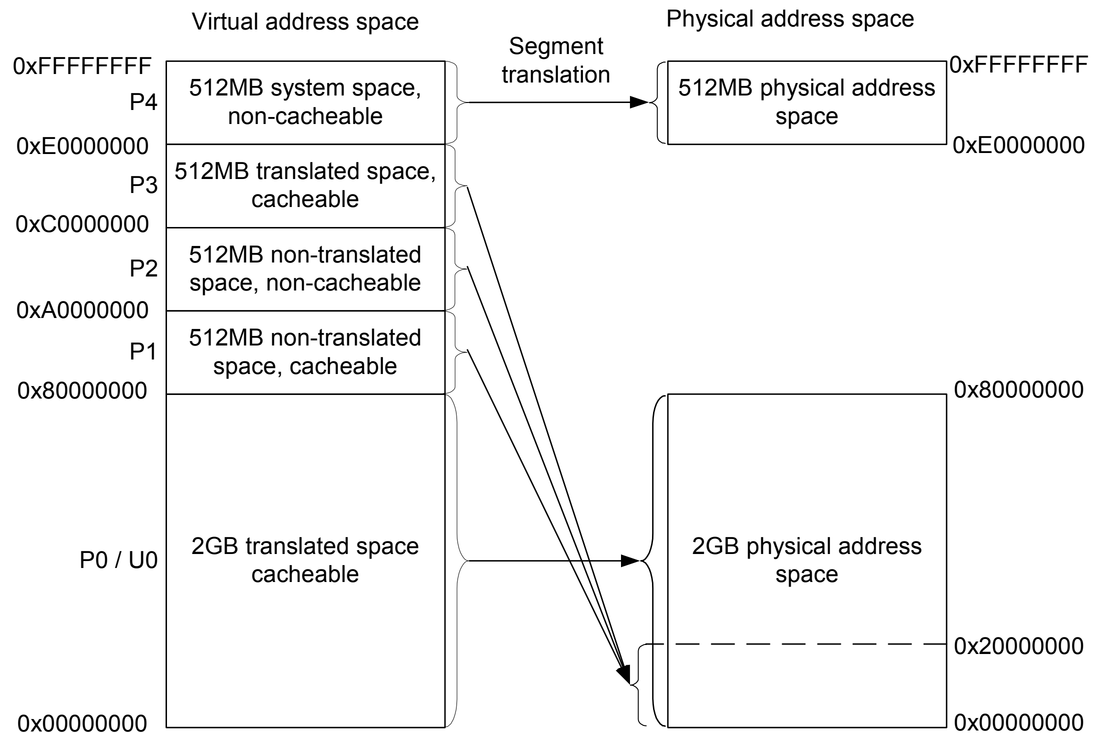
Both these translations are performed by the MMU and they can be applied independent of each other. This means that you can enable:

1. No translation. Virtual and physical addresses are the same.
2. Segment translation only. The virtual and physical addresses are the same for addresses residing in the P0, P4 and U0 segments. P1, P2 and P3 are mapped to the physical address range 0x00000000 to 0x1FFFFFFF.
3. Page translation only. All addresses are mapped as described by the TLB entries. Doing this will give all access permission control to the AP bits in the TLB entry matching the virtual address, and allow all virtual addresses to be translated.
4. Both segment and page translations. P1 and P2 are mapped to the physical address range 0x00000000 to 0x1FFFFFFF. U0, P0 and P3 are mapped as described by the TLB entries. The virtual and physical addresses are the same for addresses residing in the P4 segment.

The segment translation is by default turned on and the page translation is by default turned off after reset. The segment translation is summarized in Figure 4-2 on page 50.

www.BDTIC.com/ATMEL

Figure 4-2. The AVR32 segment translation map



4.2 Understanding the MMU

The AVR32 Memory Management Unit (MMU) is responsible for mapping virtual to physical addresses. When a memory access is performed, the MMU translates the virtual address specified into a physical address, while checking the access permissions. If an error occurs in the translation process, or operating system intervention is needed for some reason, the MMU will issue an exception, allowing the problem to be resolved by software.

The MMU architecture uses paging to map memory pages from the 32-bit virtual address space to a 32-bit physical address space. Page sizes of 1, 4, 64 Kbytes and 1 Mbyte are supported. Each page has individual access rights, providing fine protection granularity.

The information needed in order to perform the virtual-to-physical mapping resides in a page table. Each page has its own entry in the page table. The page table also contains protection information and other data needed in the translation process. Conceptually, the page table is accessed for every memory access, in order to read the mapping information for each page.

4.2.1 Virtual Memory Models

The MMU provides two different virtual memory models, selected by the Mode (M) bit in the MMU Control Register:

- Shared virtual memory, where the same virtual address space is shared between all processes
- Private virtual memory, where each process has its own virtual address space

In shared virtual memory, the virtual address uniquely identifies which physical address it should be mapped to. Two different processes addressing the same virtual address will always access

www.BDTIC.com/ATMEL

the same physical address. In other words, the Virtual Page Number (VPN) section of the virtual address uniquely specifies the Physical Frame Number (PFN) section in the physical address.

In private virtual memory, each process has its own virtual memory space. This is implemented by using both the VPN and the Application Space Identifier (ASID) of the current process when searching the TLB for a match. Each process has a unique ASID. Therefore, two different processes accessing the same VPN won't hit the same TLB entry, since their ASID is different. Pages can be shared between processes in private virtual mode by setting the Global (G) bit in the page table entry. This will disable the ASID check in the TLB search, causing the VPN section uniquely to identify the PFN for the particular page.

4.2.2 MMU interface registers

The following registers are used to control the MMU, and provide the interface between the MMU and the operating system. Most registers can be altered both by the application software (by writing to them) and by hardware when an exception occurs. All the registers are mapped into the System Register space, their addresses are presented in Section 2.5 "System registers" on page 10. The MMU interface registers are shown in Figure 4-3 on page 52.

www.BDTIC.com/ATMEL

Figure 4-3. The MMU interface registers

TLBEHI

31–10	9	8	7–0
VPN	V	I	ASID

TLBELO

31–10	9	8	7	6–5	4–3	2	1	0
PFN	C	G	B	AP	SZ	D	W	

PTBR

31–0
PTBR

TLBEAR

31–0
TLBEAR

MMUCR

31–26	25–20	19	18–14	13	12–8	7–5	4	3	2	1	0
-	-	-	DRP	-	DLA	-	S	N	I	M	E

TLBARLO

31–0
TLBARLO

4.2.2.1 *TLB Entry Register High Part - TLBEHI*

The content of the TLBEHI and TLBELO registers is loaded into the TLB when the *tlbw* instruction is executed. The TLBEHI register consists of the following fields:

- VPN - Virtual Page Number in the TLB entry. This field contains 22 bits, but the number of bits used depends on the page size. A page size of 1 Kb requires 22 bits, while larger page sizes require fewer bits. When preparing to write an entry into the TLB, the virtual page number of the entry to write should be written into VPN. When an MMU-related exception has occurred, the virtual page number of the failing address is written to VPN by hardware.
- V - Valid. Set if the TLB entry is valid, cleared otherwise. This bit is written to 0 by a reset. If an access to a page which is marked as invalid is attempted, an TLB Miss exception is raised. Valid is set automatically by hardware whenever an MMU exception occurs.
- I - Instruction TLB. The I bit is set by hardware when an MMU-related exception occurs, indicating whether the error was caused by instructions or data. All MMU operations always use the unified TLB no matter which state the I bit is in.
- ASID - Application Space Identifier. The operating system allocates a unique ASID to each process. This ASID is written into TLBEHI by the OS, and used in the TLB address match if the MMU is running in Private Virtual Memory mode and the G bit of the TLB entry is cleared. ASID is never changed by hardware.

www.BDTIC.com/ATMEL

4.2.2.2 *TLB Entry Register Low Part - TLBELO*

The content of the TLBEHI and TLBELO registers is loaded into the TLB when the *tlbw* instruction is executed. None of the fields in TLBELO are altered by hardware. The TLBELO register consists of the following fields:

- PFN - Physical Frame Number to which the VPN is mapped. This field contains 22 bits, but the number of bits used depends on the page size. A page size of 1 Kb requires 22 bits, while larger page sizes require fewer bits. When preparing to write an entry into the TLB, the physical frame number of the entry to write should be written into PFN.
- C - Cacheable. Set if the page is cacheable, cleared otherwise.
- G - Global bit used in the address comparison in the TLB lookup. If the MMU is operating in the Private Virtual Memory mode and the G bit is set, the ASID won't be used in the TLB lookup.
- B - Bufferable. Set if the page is bufferable, cleared otherwise.
- AP - Access permissions specifying the privilege requirements to access the page. The following permissions can be set, see Table 4-2:

Table 4-2. Access permissions implied by the AP bits

AP[2:0]	Privileged mode	Unprivileged mode
000	Read	None
001	Read / Execute	None
010	Read / Write	None
011	Read / Write / Execute	None
100	Read	Read
101	Read / Execute	Read / Execute
110	Read / Write	Read / Write
111	Read / Write / Execute	Read / Write / Execute

- SZ - Size of the page. The following page sizes are provided, see Table 4-3:

Table 4-3. Page sizes implied by the SZ bits

SZ[1:0]	Page size	Bits used in VPN	Bits used in PFN
00	1 Kb	TLBEHI[31:10]	TLBELO[31:10]
01	4 Kb	TLBEHI[31:12]	TLBELO[31:12]
10	64 Kb	TLBEHI[31:16]	TLBELO[31:16]
11	1 Mb	TLBEHI[31:20]	TLBELO[31:20]

- D - Dirty bit. Set if the page has been written to, cleared otherwise. If the memory access is a store and the D bit is cleared, an Initial Page Write exception is raised.
- W - Write through. If set, a write-through cache update policy should be used. Write-back should be used otherwise. The bit is ignored if the cache only supports write-through or write-back.

www.BDTIC.com/ATMEL

4.2.2.3 *Page Table Base Register - PTBR*

This register points to the start of the page table structure. The register is not used by hardware, and can only be modified by software. The register is meant to be used by the MMU-related exception routines.

4.2.2.4 *TLB Exception Address Register - TLBEAR*

This register contains the virtual address that caused the most recent MMU-related exception. The register is updated by hardware when such an exception occurs.

4.2.2.5 *MMU Control Register - MMUCR*

The MMUCR controls the operation of the MMU. The MMUCR has the following fields:

- DRP - Data TLB Replacement Pointer. DRP points to the TLB entry to overwrite when a new entry is loaded by the *tlbw* instruction. The DRP field is incremented automatically by hardware upon every *tlbw* instruction. If DRP wraps around after such an incrementation, DRP is set to the value indicated by DLA. The DRP field can also be written by software, allowing the exception routine to implement a replacement algorithm in software. The DRP field is 5 bits wide, to support 32 entries in the UTLB.
  When a DTLB protection exception, DTLB modified exception, or ITLB protection exception occurs on a valid page, the DRP is set to the index of that page.
- DLA - Data TLB Lockdown Amount. Specified the number of locked down TLB entries. All TLB entries from entry 0 to entry (DLA-1) are locked down. If DLA equals zero, no entries are locked down. A DLA setting does not prevent the programmer from modifying an entry in the TLB. DLA is only used when the *tlbw* autoincrement of DRP causes DRP to wrap.
- S - Segmentation Enable. If set, the segmented memory model is used in the translation process. If cleared, the memory is regarded as unsegmented. The S bit is set after reset.
- N - Not Found. Set if the entry searched for by the TLB Search instruction (*tlbs*) was not found in the TLB.
- I - Invalidate. Writing this bit to one invalidates all TLB entries. The bit is always read as zero.
- M - Mode. Selects whether the shared virtual memory mode or the private virtual memory mode should be used. The M bit determines how the TLB address comparison should be performed, see Table 4-4.

Table 4-4. MMU mode implied by the M bit

M	Mode
0	Private Virtual Memory
1	Shared Virtual Memory

- E - Enable. If set, the MMU page translation is enabled. If cleared, no page translation is performed.

4.2.2.6 *TLB Accessed Register HI - TLBARHI*

TLBARHI is not implemented since only 32 TLB entries are present.

www.BDTIC.com/ATMEL

4.2.2.7 TLB Accessed Register LO - TLBARLO

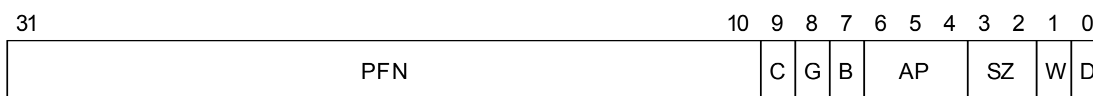
The TLBARLO register is a 32-bit register with 32 1-bit fields. Each of these fields contain the Accessed bit for the corresponding UTLB entry. Bit 0 in TLBARLO correspond to UTLB entry 0, bit 31 in TLBARLO correspond to UTLB entry 32.

Note: The contents of TLBARLO are reversed to let the Count Leading Zero (CLZ) instruction be used directly on the contents of the registers. E.g. if CLZ returns the value four on the contents of TLBARLO, then item four is the first unused item in the TLB.

4.2.3 Page Table Organization

The MMU leaves the page table organization up to the OS software. Since the page table handling and TLB handling is done in software, the OS is free to implement different page table organizations. It is recommended, however, that the page table entries (PTEs) are of the format shown in Figure 4-4. This allows the loaded PTE to be written directly into TLBELO, without the need for reformatting. How the PTEs are indexed and organized in memory is left to the OS.

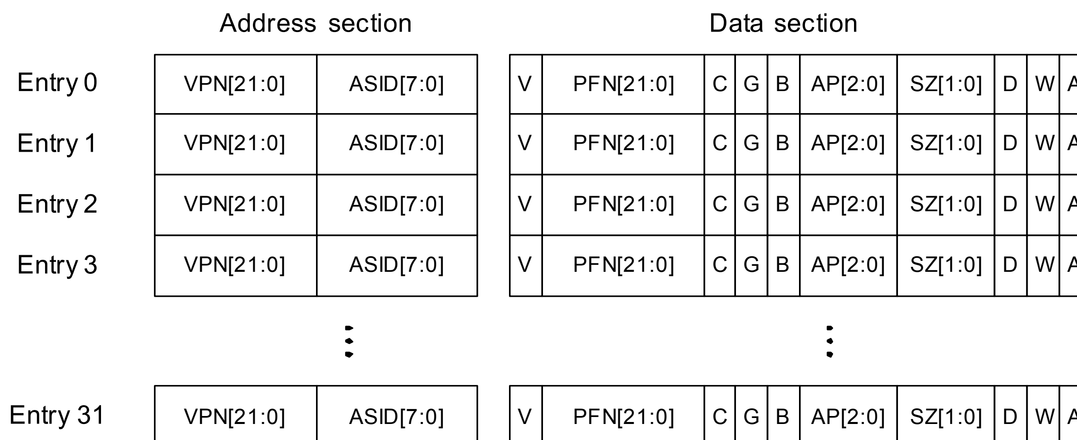
Figure 4-4. Recommended Page Table Entry format



4.2.4 TLB organization

The TLB is used as a cache for the page table, in order to speed up the virtual memory translation process. A single TLB is implemented in AVR32 AP, with 32 entries. The TLB is configured as shown in Table 4-5.

Figure 4-5. TLB organization



The A bit is the Accessed bit. This bit is set when the TLB entry is loaded with a new value using the *tlbw* instruction. It is cleared whenever the TLB matching process finds a match in the specific TLB entry. The A bit is used to implement pseudo-LRU replacement algorithms.

When an address look-up is performed by the TLB, the address section is searched for an entry matching the virtual address to be accessed. The matching process is described in chapter 4.2.5.

www.BDTIC.com/ATMEL

The MMU has a 4-entry micro-ITLB, and an 8 entry micro-DTLB connected to the caches. The caches use the micro-TLBs directly for look-ups. If the desired entry is not found in the small micro-TLB, the larger common TLB is searched. If the entry is found in the common TLB, it is copied into the desired micro-TLB and the access is performed. Otherwise, a page miss exception is issued.

The use of micro-TLBs is completely transparent to the user. Hardware is responsible for replacing entries in the micro-TLB with entries found in the main TLB. Small micro-TLBs are used in order to increase clock frequency, since performing a look-up in a large TLB is slower than for a small TLB. If an access misses in the micro-TLB, a clock cycle penalty is imposed for performing a look-up in the large TLB.

4.2.5 Translation process

The translation process maps addresses from the virtual address space to the physical address space. The addresses are generated as shown in Table 4-5, depending on the page size chosen:

Table 4-5. Physical address generation

Page size	Physical address
1 Kb	PFN[31:10], VA[9:0]
4 Kb	PFN[31:12], VA[11:0]
64 Kb	PFN[31:16], VA[15:0]
1 Mb	PFN[31:20], VA[19:0]

www.BDTIC.com/ATMEL

A data memory access can be described as shown in Table 4-6.

Table 4-6. Data memory access pseudo-code example

```
If (Segmentation disabled)
  If (! PagingEnabled)
    PerformAccess(cached, write-back);
  else
    PerformPagedAccess(VA);
else
  if (VA in Privileged space)
    if (InApplicationMode)
      SignalException(DTLB Protection, accesstype);
  endif;

  if (VA in P4 space)
    PerformAccess(non-cached);
  else if (VA in P2 space)
    PerformAccess(non-cached);
  else if (VA in P1 space)
    PerformAccess(cached, writeback);
  else
    // VA in P0, U0 or P3 space
    if ( ! PagingEnabled)
      PerformAccess(cached, writeback);
    else
      PerformPagedAccess(VA);
    endif;
  endif;
endif;
```


www.BDTIC.com/ATMEL

The translation process performed by PerformPagedAccess( ) can be described as shown in Table 4-7.

Table 4-7. PerformPagedAccess( ) pseudo-code example

```
match ← 0;
for (i=0; i<TLBentries; i++)
  if ( Compare(TLB[i]_VPN, VA, TLB[i]_SZ, TLB[i]_V) )
    // VPN and VA matches for the given page size and entry valid
    if ( SharedVirtualMemoryMode or
      (PrivateVirtualMemoryMode and ( TLB[i]_G or (TLB[i]_ASID==TLBEHI_ASID) ) ) )
        if (match == 1)
          SignalException(TLBmultipleHit);
        else
          match ← 1;
          TLB[i]_A  ← 1;
          ptr ← i;
          // pointer points to the matching TLB entry
        endif;
    endif;
endfor;

if (match == 0 )
  SignalException(DTLBmiss, accesstype);
endif;

if (InApplicationMode)
  if (TLB[ptr]_AP[2] == 0)
    SignalException(DTLBprotection, accesstype);
  endif;
endif;

if (accesstype == write)
  if (TLB[ptr]_AP[1] == 0)
    SignalException(DTLBprotection, accesstype);
  endif;
  if (TLB[ptr]_D == 0)
    // Initial page write
    SignalException(DTLBmodified);
  endif;
endif;

if (TLB[ptr]_C == 1)
  if (TLB[ptr]_W == 1)
    PerformAccess(cached, write-through);
  else
    PerformAccess(cached, write-back);
  endif;
else
  PerformAccess(non-cached);
endif;
```


www.BDTIC.com/ATMEL

An instruction memory access can be described as shown in Table 4-8.

Table 4-8. Instruction memory access pseudo-code example

```
If (Segmentation disabled)
  If (! PagingEnabled)
    PerformAccess(cached, write-back);
  else
    PerformPagedAccess(VA);
else
  if (VA in Privileged space)
    if (InApplicationMode)
      SignalException(ITLB Protection, accesstype);
  endif;

  if (VA in P4 space)
    PerformAccess(non-cached);
  else if (VA in P2 space)
    PerformAccess(non-cached);
  else if (VA in P1 space)
    PerformAccess(cached, writeback);
  else
    // VA in P0, U0 or P3 space
    if ( ! PagingEnabled)
      PerformAccess(cached, writeback);
    else
      PerformPagedAccess(VA);
    endif;
  endif;
endif;
```


www.BDTIC.com/ATMEL

The translation process performed by PerformPagedAccess( ) can be described as as shown in Table 4-9.

Table 4-9. PerformPagedAccess( ) pseudo-code example

```
match ← 0;
for (i=0; i<TLBentries; i++)
  if ( Compare(TLB[i]VPN, VA, TLB[i]SZ, TLB[i]V) )
    // VPN and VA matches for the given page size and entry valid
    if ( SharedVirtualMemoryMode or
      (PrivateVirtualMemoryMode and ( TLB[i]G or (TLB[i]ASID==TLBEHIASID) ) ) )
        if (match == 1)
          SignalException(TLBmultipleHit);
        else
          match ← 1;
          TLB[i]A  ← 1;
          ptr ← i;
          // pointer points to the matching TLB entry
        endif;
    endif;
endfor;

if (match == 0 )
  SignalException(ITLBmiss);
endif;

if (InApplicationMode)
  if (TLB[ptr]AP[2]  == 0)
    SignalException(ITLBprotection);
  endif;
endif;

if (TLB[ptr]AP[0]  == 0)
  SignalException(ITLBprotection);
endif;

if (TLB[ptr]C  == 1)
  PerformAccess(cached);
else
  PerformAccess(non-cached);
endif;
```

4.3 Operation of the MMU and MMU exceptions

The MMU uses both hardware and software mechanisms in order to perform its memory remapping operations. The following tasks are performed by hardware:

1. The MMU decodes the virtual address and tries to find a matching entry in the TLB. This entry is used to generate a physical address. If no matching entry is found, a TLB miss exception is issued.
2. The matching entry is used to determine whether the access has the appropriate access rights, cacheability, bufferability and so on. If the access is not permitted, a TLB Protection Violation exception is issued.

www.BDTIC.com/ATMEL

3. If any other event arises that requires software intervention, an appropriate exception is issued.
4. If the correct entry was found in the TLB, and the access permissions were not violated, the memory access is performed without any further software intervention.

The following tasks must be performed by software:

1. Setup of the MMU hardware by initializing the MMU-related registers and data structures if needed.
2. Maintenance of the TLB structure. TLB entries are written, invalidated and replaced by means of software. A *tlbw* instruction is included in the instruction set to support this.
3. The MMU may generate several exceptions. Software exception handlers must be written in order to service these exceptions.

4.3.1 The tlbw instruction

The *tlbw* instruction is implemented in order to aid in performing TLB maintenance. The instruction copies the contents of TLBEHI and TLBELO into the TLB entry pointed to by the DTLB Replacement Pointer (DRP) in the MMU Control Register. DRP is automatically incremented by hardware in order to implement a TLB replacement algorithm in hardware. Software may update DRP before executing *tlbw* in order to implement a software replacement algorithm.

4.3.2 TLB synonyms

The caches in the AVR32 AP system are virtually indexed but physically tagged. This allows a cache access to start before the MMU translation has completed, but puts some restrictions on which address translations can be performed.

If using pages smaller than 1/4th of the cache size, it is possible that the virtual address and the physical address could map to different places in the cache. To avoid unpredictable behaviour, the OS must ensure that no translations change address bits that are lower than 1/4th cache size.

This means that all page translation must fulfill the following restriction:

$$\text{Address}_{\text{Physical}} \text{ modulo (Cache Size / 4)} = \text{Address}_{\text{Virtual}} \text{ modulo (Cache Size / 4)}$$

For cache sizes up to 16 kB, this is only relevant for 1kB MMU pages. For 32kB caches, this is also relevant for 4kB pages.

Example 1: On a system with 8kB caches, virtual and physical address must be the same modulo 2 kB. If using 1kB pages, the OS must ensure that bit 10 of the address is not changed by the translation.

Example 2: On a system with 16kB caches, virtual and physical address must be the same modulo 4 kB. If using 1kB pages, the OS must ensure that bit 10 and 11 of the address are not changed by the translation.

4.3.3 MMU exception handling

This chapter describes the software actions that must be performed for MMU-related exceptions. The hardware actions performed by the exceptions are described in detail in Section 3.11.1 ”Description of events in AVR32 AP” on page 35.

www.BDTIC.com/ATMEL

4.3.3.1 ITLB / DTLB Multiple Hit

If multiple matching entries are found when searching the TLB, or matching entries are found in a segment translated area, this exception is issued. This situation is a critical error, since memory consistency can no longer be guaranteed. The exception hardware therefore jumps to the reset vector, where software should execute the required reset code. This exception is a sign of erroneous code and is not normally generated.

The software handler should perform a normal system restart. However, debugging code may be inserted in the handler.

4.3.3.2 ITLB / DTLB Miss

This exception is issued if no matching entries are found in the TLB, or when a matching entry is found with the Valid bit cleared.

1. Examine the TLBEAR and TLBEHI registers in order to identify the page that caused the fault. Use this to index the page table pointed to by PTBR and fetch the desired page table entry.
2. Use the fetched page table entry to update the necessary bits in TLBEHI and TLBELO. The following bits must be updated, not all bits apply to ITLB entries: V, PFN, C, G, B, AP[2:0], SZ[1:0], W, D.
3. The replacement pointer in MMUCR[DRP] may be written to manually choose which entry to replace.
4. Execute the *tlbw* instruction in order to update the TLB entry.
5. Finish the exception handling and return to the application by executing the *rete* instruction.

4.3.3.3 ITLB / DTLB Protection Violation

This exception is issued if one of the following occur:

- Access to a privileged segment in application mode.
- Access to a page translated area, and the access permissions on the matching page does not allow that type of access. MMUCR[DRP] is updated to point to the matching TLB entry.

Software must examine the TLBEAR and TLBEHI registers in order to identify the instruction and process that caused the error. Corrective measures like terminating the process must then be performed before returning to normal execution with *rete*.

4.3.3.4 DTLB Modified

This exception is issued if a valid memory write operation is performed to a page that has never been written before. This is detected by the Dirty-bit in the matching TLB entry reading zero.

1. Examine the TLBEAR and TLBEHI registers in order to identify the page that caused the fault. Use this to index the page table pointed to by PTBR and fetch the desired page table entry.
2. Set the Dirty bit in the read page table entry and write this entry back to the page table
3. Use the fetched page table entry to update the necessary bits in TLBEHI and TLBELO. The following bits must be updated: V, PFN, C, G, B, AP[2:0], SZ[1:0], W, D.
4. The TLBEHI[I] register is cleared by hardware to indicate that it was a data access, and MMUCR[DRP] is updated to point to the matching TLB entry.
5. Execute the *tlbw* instruction in order to update the TLB entry.
6. Finish the exception handling and return to the application by executing the *rete* instruction.

www.BDTIC.com/ATMEL

5. Prefetch Unit

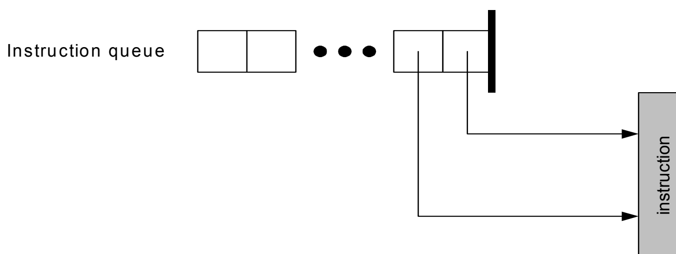
5.1 Instruction buffer

The instruction buffer is implemented as a 96-bit FIFO queue, holding 12 byte-sized entries. The buffer can hold either Java or RISC instructions. The instruction at the front of the queue is issued to the ID stage at each clock cycle. The buffer detects the length of this instruction, and shifts the queue the appropriate amount. The tail of the queue is filled with instructions from the instruction cache. Instructions are placed in the buffer as soon as the queue has vacant slots. If the queue is empty, or the instruction at the head of the queue is only partially fetched, the ID stage may need to stall until the entire instruction is available.

The queue can contain instructions of different length. The instruction at the front of the queue is always assumed to be a valid, aligned and complete instruction. If this condition fails, the hardware would be unable to determine the instruction boundary. This is necessary in order to separate between instructions and decide where in the buffer an instruction starts and ends.

The instruction buffer has the following format:

Figure 5-1. Instruction buffer.



5.1.1 Instruction buffer fill

If the instruction buffer is non-empty, fetched instructions will always reside at sequential addresses of the instructions already in the buffer. In this case, no special concerns are taken. If the instruction buffer has been flushed and is empty, the loaded word must be rotated in such a way that the addressed instruction is placed at the front of the instruction queue. If the target address is not word-aligned, the most significant bytes in the fetched word are discarded. In order to efficiently execute branch instructions, the branch targets for extended branch target instructions should be aligned. This will in many cases allow the target instruction to be fetched and executed without pipeline stalls.

A dedicated fetch address adder generates addresses to fetch from the instruction cache for sequential instruction flow.

5.1.2 Flushing of the instruction buffer

The instruction buffer is flushed in the following circumstances:

- Entry into an exception routine
- Execution of an instruction with PC as target register
- Excecution of an unpredicted branch
- Detection of a mispredicted branch
- A procedure return address is popped from the return address stack.

www.BDTIC.com/ATMEL

5.1.3 Instruction forwarding

If for some reason the instruction buffer is empty, the fetched instruction will be forwarded past the fetch queue and into ID as soon as it is fetched. This forwarding is done only when it can be ensured that the entire instruction is contained in the fetched word. This is ensured if the 2 least significant bits of the PC of the desired instruction equal 00. If the bits are 10, the fetched instruction must pass via the instruction buffer. This will impose a 1-cycle penalty on the case where the addressed instruction is a compact instruction, but is done for critical path simplification. Instruction forwarding is done only in RISC mode, no instruction forwarding is done in Java mode.

5.2 Branch prediction

5.2.1 Functionality

AVR32 AP implements special hardware in order to minimize the penalty from mispredicted branches. A branch target buffer (BTB) is used in order to record information about the outcome of encountered branches. This information is used to predict the outcome of branches based on the recorded history of the branch. The hardware is able to start fetching instructions from the predicted path, so that no branch penalty is experienced if the prediction was correct. If the prediction was wrong, the pipeline will have to be flushed and execution resume at the correct path.

If prediction information about an encountered branch is contained in the BTB, hardware will in many cases be able to fold the branch instruction with the following instruction. In this process, the branch instruction is removed from the execution stream and its condition codes are passed on to the following instruction. This instruction is sent down the pipe together with the condition codes of the branch. If the branch was predicted correctly, the folded instruction is allowed to complete. Otherwise, the folded instruction is flushed from the pipeline and execution continues from the alternate branch path.

Branch prediction is enabled or disabled according to the Branch Prediction Enable (BE) bit in the CPUCR system register. Before enabling or disabling the BTB, it must be invalidated. No branches should be executed between the BTB invalidate and the BTB enable or disable.

Branch prediction is invisible to the programmer. Hardware makes sure that the program executes correctly regardless of a branch being predicted or not, and the correctness of the prediction.

www.BDTIC.com/ATMEL

5.2.2 Predictable instructions

The following instructions are predictable, and may be placed in the BTB.

Table 5-1. Predictable instructions

Instruction	Conditional	Mode	Can fold other instructions	Can merge with other instructions
br k8	Yes	RISC	Yes	-
br k21	Yes	RISC	Yes	-
rjmp k10	No	RISC	Yes	-
rjmp k21	No	RISC	Yes	-
rcall k10	No	RISC	No	-
rcall k21	No	RISC	No	-
if{cond}	Yes	Java	-	Yes
ifcmp{cond}	Yes	Java	-	Yes

5.2.3 Foldable instructions

All instructions can be folded into a branch instruction, except for instructions that are predicted by the BTB. These instructions are listed in Table 5-1. In other words, *br* {k8, k21} and *rjmp* {k10, k21} can not be folded together with any of the instructions listed in Table 5-1. The three call instructions listed in the table can not fold into any other instructions.

All Java branches can be predicted and can be merged as described in the Java Technical Reference.

5.2.4 Branch target buffer

The branch target buffer (BTB) is a n-entry direct mapped cache. The indexing function used is index = fetchadr[n+2:2]. This function is expected to present a good hashing function into the cache, distributing competing entries evenly into the cache. Note that bits [n+2:2] in the instruction address is used for BTB lookup. This will map two sequential compact branches to the same BTB entry. This should not reduce performance, as the case where two sequential branches both are predicted taken is meaningless. The other bits in the instruction address is used as cache tag fields.

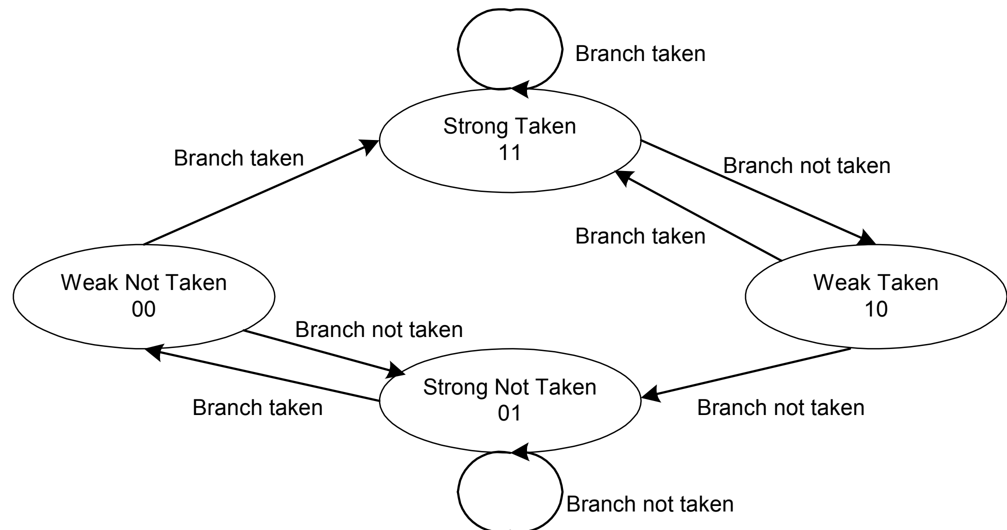
Each line in the BTB cache cache has the following format. The Ext field indicates if the branch instruction is an extended instruction.

Figure 5-2. BTB entry.

The history bits are implemented as a 2-bit saturating counter. When a new branch is detected, the counter is initialized to Strong Taken. The FSM has the following coding and transitions:

www.BDTIC.com/ATMEL

Figure 5-3. BTB entry.



5.2.5 BTB update policy

A new entry is stored in the BTB when the following conditions are met:

- The branch instruction or the branch-folded instruction has executed, i.e. left A1.
- The branch was taken
- The branch is not currently in the BTB
- Branch prediction is enabled

Once a branch is stored in the BTB, the history bits are updated upon every execution of the branch. When a new branch is detected, the counter is initialized to Strong Taken, and the Valid bit is set.

5.2.6 Reset

Branch prediction is enabled after reset. All valid bits in the BTB are cleared after reset.

5.2.7 Invalidation

The BTB is invalidated when one or more of the following events occur:

- Reset
- The instruction cache is invalidated
- The ASID part of the TLBEHI system register is written
- The BTB Invalidate (BI) bit in the CPUCR system register

The application may manually need to invalidate the BTB after executing self-modifying code, in order to avoid false predictions.

Important note:

As mentioned above, the BTB is invalidated when the ASID field in TLBEHI is written. As shown in Table 2-2, “System Registers implemented in AVR32 AP,” on page 10, TLBEHI is accessed through the TCB bus. This implies that several instructions can be present upstream in the pipeline when TLBEHI is written. However, writing to ASID does not cause the pipeline to be flushed. The user must therefore ensure that predicted instructions is not fetched and sent down the

www.BDTIC.com/ATMEL

pipeline before the write to ASID has invalidated the BTB. Failing to do so may cause UNDEFINED behaviour. This error can be avoided by one of the following methods:

- Scheduling the code executed after the write to TLBEHI, such that no predictable instructions are fetched before the BTB has been invalidated by the write to TLBEHI, or
- Forcing a pipeline flush by issuing an unpredicted change-of-flow instruction after the write to the TLBEHI, so that the pipeline is flushed after the BTB has been invalidated. This can be done by the following code sequence:

```
mtsr AVR32_TLBEHI, r0   ; Update TLBEHI with new value present in r0
sub  pc, -2             ; Not predictable COF insn flushing the pipe
```

5.2.8 Disabling

Branch prediction is disabled by clearing the Branch Prediction Enable (BE) bit in CPUCR.

www.BDTIC.com/ATMEL

5.2.9 Return stack

The return stack is a 4-entry circular buffer, holding the return addresses for call instructions. The ID stage controls the pushing of return addresses to the return stack. When A1 detects a *rcall*, *icall*, *mcall* or *acall* instruction, the address of the instruction following the call (the instruction in IS) is pushed. This is the return address.

There are two types of return instructions: Predicted taken and predicted not taken. The prediction is statically based on the instruction opcode, as shown in Table 5-2. Predicted taken return instructions will cause a return stack pop and execution will continue as soon as possible from the new path. Predicted not taken return instructions will not cause a return stack pop until it has reached the A1 stage and the condition is evaluated to be true.

When ID detects a predicted taken return instruction, the top-of-stack element is popped and used as an instruction fetch address. The return stack is circular, and overflow is handled by hardware by means of a saturating valid-element counter. If a predicted taken return instruction is encountered and the return stack is empty, the return instruction will still be executed correctly, but with a cycle penalty.

The instructions in Table 5-2 are considered return-instructions. No other instructions or mechanisms should be used to return from call-instructions. Violation of this rule will place hardware in an undetermined state. If the user wants to return from call-instructions by other means than the instructions listed in Table 5-2, the return stack must be disabled by clearing the Return Stack Enable (RE) bit in CPUCR. Another approach is to flush the return stack before returning by executing the flush return stack (*frs*) instruction.

Table 5-2. Predictable return instructions

Instruction	Prediction
mov pc, lr	Taken
ret, cond == AL	Taken
ret, cond != AL	Not taken
popm with PC in register list	Taken
ldm with PC in register list	Taken

5.2.10 Reset

The return stack is enabled and empty after reset.

5.2.11 Disabling

The return stack is disabled by clearing the Return Stack Enable (RE) bit in CPUCR. Disabling the stack will reset the element counter, removing all entries.

www.BDTIC.com/ATMEL

6. Instruction Cache

The AVR32 AP uses an instruction cache in order to increase performance and lower power consumption. The cache has the following features:

- Virtually indexed, physically tagged.
- 4-way Set-associative.
- 32-byte line size.
- Least recently used (LRU) allocate-on-read-miss line replacements.
- Easily portable using standard single and two port RAMs.
- Lockable on a per-line basis.
- Cacheable or uncacheable operation configurable on a per-page basis through the MMU.
- All accesses are subject to MMU protection and translation checks.
- Powerful cache maintenance operations, allowing many common cache operations to be performed through a single instruction.

The number of sets in the instruction cache is specified in the CONFIG1 register described in chapter 2.6. The total cache size is (*number of sets * line size * associativity*), i.e. (128 * *number of sets*) bytes.

6.1 Behaviour

Reset invalidates all entries in the ICache.

All instruction fetches will result in an ICache lookup and will return the cached data if the requested address is found in the cache (a cache hit). If data are found in the cache, they are returned even if the memory area is marked as uncacheable. If the address is not found (a cache miss) and the address is in a cacheable area, a burst read access is started on the system bus in order to read an entire cache line of data. The read data will be written to an unlocked line according to a LRU scheme, possibly replacing another line.

If a cache miss is in an uncacheable area, a single non-sequential read is started on the system bus.

If the MMU is disabled or the fetch is from an unmapped segment, the cacheability of memory areas is predefined as shown in the architecture manual. If the MMU is enabled and the fetch is from a mapped segment, the cacheability of memory areas is controlled through the C bit in the TLB.

If the MMU signals an exception, the cache will abandon the fetch. The exception is handled by the CPU as soon as it knows if the instruction should really be executed, but is ignored if it turns out to be a needlessly prefetched instruction.

There is no hardware support for self-modifying code. If any memory that *may* be cached in the ICache is modified during execution, the programmer is responsible for ensuring ICache consistency. See chapter 6.3 for details on how to do this.

www.BDTIC.com/ATMEL

6.2 Cache operations

All cache operations are initiated through the CACHE instruction. See the Instruction Set Description for the format of the CACHE instruction.

The following cache operations are defined for the ICache:

Table 6-1. ICache operations

Op[4:3]	Op[2:0]	Operation	Parameter
00	000	Flush	Flush mode
00	001	Invalidate	Virtual Address
00	010	Lock	Virtual Address
00	011	Unlock	Virtual Address
00	100	Prefetch	Virtual Address
00	101	Reserved	N/A
00	110	Reserved	N/A
00	111	Reserved	N/A
Other	xxx	Reserved for other caches	N/A

6.2.1 The Flush operation

The flush operation is used to reset the contents of the cache. This is done automatically at reset, and may be used at the programmers discretion at other times. The parameter is used to select one of the following flush modes:

Table 6-2. ICache flush modes

Mode	Name	Description
0	Flush all	All lines are invalidated and unlocked, including locked ones.
1	Flush unlocked	Invalidate all unlocked lines.
2	Unlock all	All locked lines are unlocked, but no lines are invalidated.
Others	Undefined	Should not be used - operation is undefined.

6.2.2 The Invalidate operation

The invalidate operation will try to invalidate the line containing the address given by the parameter. The address is treated as a virtual address, and is translated to a physical address by the MMU. If the line exists in the cache, it is marked as invalid and unlocked. Otherwise nothing is done. If any MMU exceptions happen, the operation is silently aborted.

6.2.3 The Lock operation

The lock operation will try to lock the line containing the address given by the parameter into the cache. The address is treated as a virtual address, and is translated to a physical address by the MMU. If the line exists in the cache, the lock bit is set. If the line does not exist in the cache, it will be fetched from the bus and locked - even if the area is uncacheable. If any exceptions happen - MMU or bus exceptions - the operation is silently aborted. If the requested address maps to a set with four lines already locked, the operation is silently aborted.

www.BDTIC.com/ATMEL

6.2.4 The Unlock operation

The unlock operation will try to unlock the line containing the address given by the parameter. The address is treated as a virtual address, and is translated to a physical address by the MMU. If the line exists in the cache and is locked, it is unlocked. Otherwise nothing is done. If any MMU exceptions happen, the operation is silently aborted.

6.2.5 The Prefetch operation

The prefetch operation will try to load the line containing the address given by the parameter into the cache. The address is treated as a virtual address, and is translated to a physical address by the MMU. If the line exists in the cache, nothing is done. If the line does not exist in the cache, it will be fetched from the bus - even if the area is uncacheable. If any exceptions happen - MMU or bus exceptions - the operation is silently aborted.

6.3 Memory coherency

Whenever code is modified in some way, e.g. through self-modifying code, there is a chance that the instruction cache may hold cached copies of the old instructions. To ensure correct execution of the new code, the user must manually force the caches to update. The procedures for doing so are described below.

6.3.1 DMA of program code

If some peripheral updates program code through DMA to memory, the instruction cache may hold cached copies of the old code. The old code must be manually flushed from the instruction cache by following these steps:

1. Flush the entire instruction cache, as described in chapter 6.2.1, or
2. Flush only the affected memory areas through one or more Invalidate operations, as described in chapter 6.2.2.
3. Jump to the new code using an unpredicted branch

Example:

```
mov R0, 0
cache ICACHE_FLUSH, R0[ICACHE_INVALIDATE_ALL]
mov PC, new_code_label
```

6.3.2 Self-modifying code

If the CPU updates the code through writing to memory, the updated code may be buffered in the data cache or the write buffer, and the instruction cache may hold cached copies of the old code. The caches must be manually updated by following these steps:

1. Clean the entire data cache, as described in TODO, or
2. Clean only the affected memory areas through one or more Clean operations, as described in TODO.
3. Empty the write buffer, as described in TODO
4. Flush the entire instruction cache, as described in chapter 6.2.1, or
5. Flush only the affected memory areas through one or more Invalidate operations, as described in chapter 6.2.2.
6. Jump to the new code using an unpredicted branch

www.BDTIC.com/ATMEL

Example:

```
Mov R0, 0
cache DCACHE_FLUSH, R0[DCACHE_CLEAN_ALL]
sync 0
cache ICACHE_FLUSH, R0[ICACHE_INVALIDATE_ALL]
mov PC, new_code_label
```

6.4 Debug access to ICache memories

It is possible to directly access the memories in the ICache through the SAB bus.

The ICache maps read or write requests to the cache memories by decoding the address as shown below:

Figure 6-1. ICache direct access addressing.

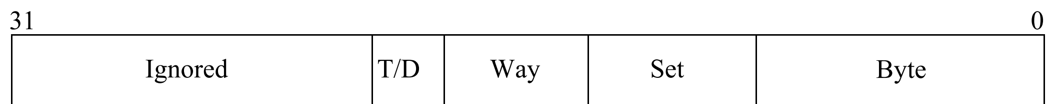


Table 6-3. ICache direct access address fields

Field	Size (bits)	Description
T/D	1	Access tag memories if set, data memories if cleared
Way	2	Selects which line in a set to access.
Set	log2(number of sets)	Selects which set to access.
Byte	5	Selects which byte of the line to access. Ignored if accessing tag memory.

Note that the ICache only supports word-aligned 32-bit accesses. Other sizes or alignments may cause undefined behaviour.

www.BDTIC.com/ATMEL

6.4.1 Format of tag memory

Each word in the tag memory is formatted as shown below:

Figure 6-2. ICache direct access tag format.

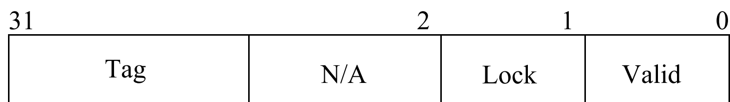


Table 6-4. ICache direct access tag fields

Field	Size (bits)	Description
Tag	27 - log2(number of sets)	The most significant bits of the physical address of the data in the specified line.
N/A	30 - tag size	Not used. Reads are undefined, writes are ignored.
Lock	1	Set if the line is locked, cleared otherwise.
Valid	1	Set if the line is valid, cleared otherwise.

www.BDTIC.com/ATMEL

7. Data Cache and Write Buffer

The AVR32 AP uses a data cache and a write buffer in order to increase performance and reduce power consumption. The data cache has the following features:

- Virtually indexed, physically tagged.
- 4-way set-associative.
- 32-byte line size
- Least recently used allocate-on-read-miss line replacements.
- Synthesizable design using standard single and two port RAMs.
- Lockable on a per-line basis.
- Cacheable or uncacheable operation configurable on a per-page basis through the MMU.
- Write-back or write-through operation configurable on a per-page basis through the MMU.
- All accesses are subject to MMU protection and translation checks.
- Powerful cache maintenance operations, allowing many common cache operations to be performed through a single instruction.

The number of sets in the data cache is specified in the CONFIG1 register described in chapter 2.6. The total cache size is (*number of sets * line size * associativity*), i.e. (128 * *number of sets*) bytes.

The write buffer has the following features:

- Bufferable or unbufferable writes configurable on a per-page basis through the MMU.
- Does write combining on bufferable writes.
- Holds up to 32 bytes of buffered data, plus up to 32 bytes of data that are about to be written to the bus.
- Can be manually flushed by using the SYNC instruction.

7.1 Data cache behaviour

Reset invalidates all entries in the DCache.

All reads result in an DCache lookup and will return the cached data if the requested address is found in the cache (a cache hit). If the address is not found (a cache miss) and the address is in a cacheable area, a burst read access is started on the system bus in order to read an entire cache line of data. The read data will be written to an unlocked line according to a round-robin scheme, possibly replacing another line.

If the cache miss is in an uncacheable area, a single non-sequential read is started on the system bus.

All writes result in a DCache lookup and will update the cached data if the requested address is found in the cache (a cache hit). If the address is not found (a cache miss) or the address is configured as write-through, the write will also update the write buffer (if bufferable) or be written to the bus (if unbufferable).

If the MMU is disabled or the fetch is from an unmapped segment, the cacheability, bufferability and write-back/write-through attributes of memory areas are predefined as shown in the architecture manual. If the MMU is enabled and the access is to a mapped segment, the attributes are configured through the C, B and W bits in the TLB.

www.BDTIC.com/ATMEL

7.2 Write buffer behaviour

Reset invalidates the entire write buffer.

Writes are separated into "write now" (unbufferable) and "write later" (bufferable) writes. Multiple bufferable writes to the same aligned 32-byte line may be merged for performance, and reads may forward buffered data directecly from the buffer. Bufferable writes will stay in the write buffer until forced out when one of the following occur:

A buffered write is to another 32-byte line than the one currently in the buffer. The old data are moved to the "write now" part of the buffer, and the new data are kept as "write later".

A read finds only part of the requested data in the write buffer, e.g a word read finds *one* byte of the word in the write buffer. The buffered data are moved to the "write now" part of the buffer, and the read is qued until the write has completed. This ensures data consistency.

A SYNC instruction is executed. All data in the write buffer are written to the bus.

Unbufferable writes are put in the "write now" part of the buffer. These are written to the bus at the earliest oportunity, in the same order as issued, and without any write combining. These data cannot be forwarded by reads, and are always performed before any read misses.

7.3 Cache and write buffer operations

7.3.1 The Cache instruction

The cache instruction can be used to send special commands to the cache with a 32-bit parameter. Se the architecture reference manual for description of the instruction. The following commands are currently defined:

Table 7-1. Cache instruction parameters

Op[4:3]	Op[2:0]	Operation	Parameter
01	000	Flush	Flush Mode
01	001	Lock	Virtual Address
01	010	Unlock	Virtual Address
01	011	Invalidate	Virtual Address
01	100	Clean	Virtual Address
01	101	Clean & Invalidate	Virtual Address
01	110	Reserved for future use	Undefined
01	111	Reserved for future use	Undefined
Other	xxx	Reserved for other caches	N/A

www.BDTIC.com/ATMEL

7.3.2 Flush

The flush operation is used to reset the contents of the cache. While a flush is active, all other operations are stalled. The parameter is used to select one of the following flush modes:

Table 7-2. Flush modes

Mode	Name	Description
0	Invalidate all	All lines are invalidated and unlocked. Dirty lines are *not* cleaned!
1	Invalidate unlocked	All unlocked lines are invalidated. Dirty lines are *not* cleaned!
2	Clean all	All lines are cleaned, but no lines are invalidated.
3	Clean Unlocked	All unlocked lines are cleaned, but no lines are invalidated.
4	Clean & Invalidate all	All lines are cleaned, invalidated and unlocked.
5	Clean & Invalidate unlocked	All unlocked lines are cleaned and invalidated.
6	Unlock all	All locked lines are unlocked.
Others	Undefined	Should not be used - operation is undefined.

7.3.3 Lock

The lock operation will try to lock the line containing the address given by the parameter into the cache. If the line exists in the cache, the lock bit is set. If the line does not exist in the cache, it will be fetched from the bus and locked - even if the area is uncacheable. If any MMU exceptions happen, the operation is silently aborted. If the requested address maps to a set with four lines already locked, the operation is silently aborted.

7.3.4 Unlock

The unlock operation will try to unlock the line containing the address given by the parameter. If the line exists in the cache and is locked, it is unlocked. Otherwise nothing is done. If any MMU exceptions happen, the operation is silently aborted.

If the line referred to is still being fetched from the bus, the unlock operation will complete but the line could be locked after the unlock completes. This is only possible if a *lock* is closely followed by an *unlock* to the same line - in this case use the *sync* instruction to make sure any pending locks are completed before the unlock is performed.

7.3.5 Invalidate

The invalidate operation will try to invalidate the line containing the address given by the parameter. If the line exists in the cache, it is marked as invalid. Otherwise nothing is done. If the line has dirty data, these updates will be lost! If any MMU exceptions happen, the operation is silently aborted.

If the line referred to is still being fetched from the bus, the invalidate operation will complete but the line could become valid after the invalidate completes. This is only possible if a read or prefetch is closely followed by an *invalidate* to the same line - in this case use the *sync* instruction to make sure any pending reads are completed before the invalidate is performed.

7.3.6 Clean

The invalidate operation will try to clean the line containing the address given by the parameter. If the line exists in the cache and has dirty data, it will be written to the write buffer and marked

www.BDTIC.com/ATMEL

as clean. If the write buffer is full the instruction will stall. If the line is not found or the line is clean, nothing is done. If any MMU exceptions happen, the operation is silently aborted.

7.3.7 Clean & Invalidate

This operation performs a *clean* (as described in chapter 7.3.6) and then a *invalidate* (as described in chapter 7.3.5).

7.4 Prefetch instruction

The prefetch operation will try to load the line containing the address given by the parameter into the cache. If the line exists in the cache, nothing is done. If the line does not exist in the cache, it will be fetched from the bus - even if the area is uncacheable. If any MMU exceptions happen, the operation is silently aborted.

7.5 Sync instructions

The sync instruction will flush the write buffer, by forcing it to write any dirty data to the bus. This can be used to ensure the data in main memory is consistent.

It will also ensure all pending read operations are completed, so e.g. *invalidate* and *unlock* operations are guaranteed to complete correctly.

7.6 Memory mapped cache memories

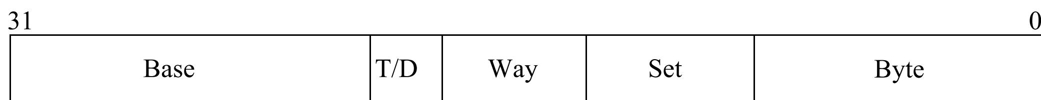
Both the data and tag memories of the DCache are mapped into the global address space, and can be accessed by programs or through the SAP or OCD system. This can be used for complex cache control, or simply as a very fast scratch RAM. The base address of the memory map is IMPLEMENTATION DEFINED, but will always be in an unmapped privileged segment. The size of the memory mapped area is always twice the cache size.

This mapped area is not available through the instruction cache or from peripherals, so it is not possible to run programs from this area, or DMA data to or from it.

Note: Incorrect values written to the memory mapped cache memories may cause data corruption or unpredictable behaviour.

The DCache maps read or write requests to the cache memories by decoding the address as shown below:

Figure 7-1. DCache direct access addressing.



www.BDTIC.com/ATMEL

Table 7-3. DCache direct access address fields

Field	Size (bits)	Description
Base	26 - log2(number of sets)	Base address of the memory mapped area. Memory mapped cache access is only enabled if this field matches the IMPLEMENTATION DEFINED base address.
T/D	1	Access tag memories if set, data memories if cleared
Way	2	Selects which line in a set to access.
Set	log2(number of sets)	Selects which set to access. Number of sets is (cache size / (associativity * line size))
Byte	5	Selects which byte of the line to access.

The DCache data memories supports any access that is aligned to the size of the access. The tag memories only supports aligned 32-bit accesses, other accesses are undefined.

7.6.1 Format of tag memory

In the tag part of the memory mapped area the following data can be found for each line:

Table 7-4. Memory map tag layout

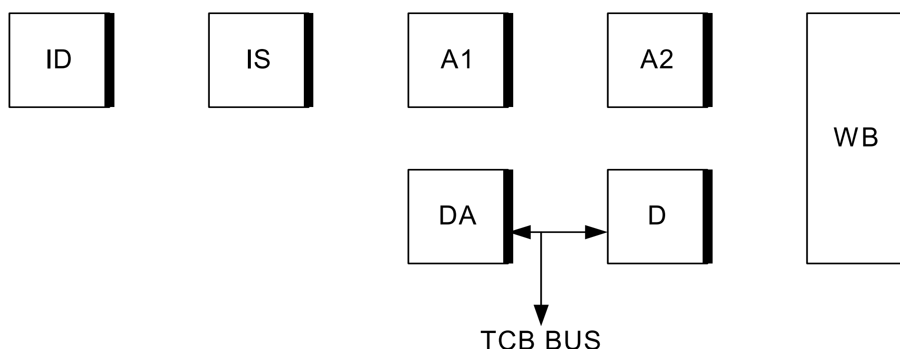
Word	Data
0	Address, lock and valid bits. The valid bit is bit 0, the lock bit is bit 1, the address bits are the upper *n* bits where *n* is 27 - log2(number of sets). The remaining bits read as zero, and are ignored on writes.
1	Dirty bits. The lower 8 bits contain one dirty bit per word of data. The word at address zero correspond to bit zero. The remaining bits read as zero, and are ignored on writes.
2	Empty. Read as zero, and is ignored on writes.
3	Replace data. The lower 6 bits contain the replace data used by the cache for selecting which line to replace within the set. This data is the same for all lines in a set. Bits 5:4 hold the index of the most recently used line within the set. Bits 3:2 hold the index of the second most recently used line within the set. Bits 1:0 hold the index of the third recently used line within the set. The least used line within the set, i.e. the one selected for replacement, is the one not listed in the above fields. The remaining bits read as zero, and are ignored on writes. **Note:** Setting two or more of the above fields to the same index may lead to unpredictable behaviour.
> 3	Words 0-3 are repeated over the entire line.

www.BDTIC.com/ATMEL

8. Coprocessor interface

The coprocessor interface allows custom peripherals such as graphics coprocessors to be tightly coupled to the CPU. No hazard detection is performed on coprocessor registers, so software must schedule instructions with care so that no hazards exist.

Figure 8-1. The coprocessor pipeline.



Command, operand and address passing to coprocessors is done via a dedicated, pipelined bus. This bus is called the Tightly Coupled Bus (TCB), and is used by the system register interface as well. Using a bus allows for easy attachment of coprocessors. The simple synchronization between the coprocessor and the CPU allows the coprocessor clock frequency to differ from that of the CPU.

The TCB bus is also used for transporting data to and from the external system registers. Accesses to these registers can be performed by placing an opcode on tcb_cmd, as done for *tlbr*, *tlbw* and *tlbs*.

8.1 Coprocessor pipeline

The coprocessor interface does not specify any special construction or architecture of the coprocessor pipelines. A coprocessor only needs to comply to the rules of the TCB. Special handshaking signals are implemented so that the coprocessor can stall the CPU pipeline if it is unable to reply to a request from the CPU.

8.2 TCB specification

The TCB bus is implemented as a pipelined bus in order to achieve maximum performance. Additionaly, handshaking has been implemented so that slow coprocessors can insert the required number of wait states. The CPU is the single master on the TCB bus, and all coprocessors are slaves. The coprocessors will only respond to transactions issued by the master, and can never initiate a transfer.

Since the TCB is a pipelined bus, a bus transaction consists of an address phase and a data transfer phase.

www.BDTIC.com/ATMEL

Table 8-1. TCB signals

Name	Dir	Description		
		Value	Semantic	Comment
tcb_cmd[7:0]	Out	0	IDLE	No TCB activity
		1	write.w	CRd on tcb_cprega Data to coprocessor on tcb_wdataa
		2	write.d	CRd+1:CRd on tcb_cprega:tcb_cpregb Data to coprocessor on tcb_wdataa:tcb_wdatab
		3	read.w	CRs on tcb_cprega Data from coprocessor on tcb_rdataa
		4	read.d	CRs+1:CRs on tcb_cprega:tcb_cpregb Data from coprocessor on tcb_rdataa:tcb_rdatab
		5	mtsr	SysRegNo[3:0] on tcb_cprega SysRegNo[7:4] on tcb_cpno Data to system register on tcb_wdataa
		6	mfsr	SysRegNo[3:0] on tcb_cprega SysRegNo[7:4] on tcb_cpno Data from system register on tcb_rdataa
		7	mtdr	DebugRegNo[3:0] on tcb_cprega DebugRegNo[7:4] on tcb_cpno Data to debug register on tcb_wdataa
		8	mfdr	DebugRegNo[3:0] on tcb_cprega DebugRegNo[7:4] on tcb_cpno Data from debug register on tcb_rdataa
		9	tlbr	tcb_cpreg* and tcb_cpno not used
		10	tlbs	tcb_cpreg* and tcb_cpno not used
		11	tlbw	tcb_cpreg* and tcb_cpno not used
		128-255	COP opcode	The opcode of the COP instruction.
tcb_cpno[3:0]	Out	Used to address coprocessors or system register blocks. 8 Different coprocessors and 16 different system register blocks are supported in AVR32 AP. Bits 2:0 are used for carrying the coprocessor number for coprocessor instructions. Bits 3:0 are used for carrying the system register block address for *mt(s,d)r* and *mf(s,d)r*, which is given by SysRegNo[7:4].		

www.BDTIC.com/ATMEL

Table 8-1. TCB signals

<table>
<tr><th>Name</th><th>Dir</th><th colspan="4">Description</th></tr>
<tr><td rowspan="9">tcb_cprega[3:0],
tcb_cpregb[3:0],
tcb_cpregc[3:0]</td><td rowspan="9">Out</td><td colspan="4">Address bus for the three operands required by the coprocessor instructions. Also used to carry part of the system register address together with tcb_cpno.</td></tr>
<tr><td>Operation</td><td>tcb_cprega</td><td>tcb_cpregb</td><td>tcb_cpregc</td></tr>
<tr><td>COP</td><td>CRd</td><td>CRx</td><td>CRy</td></tr>
<tr><td>write.w</td><td>CRd</td><td></td><td></td></tr>
<tr><td>write.d</td><td>CRd+1</td><td>CRd</td><td></td></tr>
<tr><td>read.w</td><td>CRs</td><td></td><td></td></tr>
<tr><td>read.d</td><td>CRs+1</td><td>CRs</td><td></td></tr>
<tr><td>mtsr/mtdr</td><td>SysRegNo[3:0]</td><td></td><td></td></tr>
<tr><td>mfsr/mfdr</td><td>SysRegNo[3:0]</td><td></td><td></td></tr>
<tr><td>tcb_cpuflushed</td><td>Out</td><td colspan="4">Asserted by the CPU if the CPU LS pipeline was flushed in the previous cycle. This typically occurs if a coprocessor memory operation caused an address exception in the D stage. The coprocessor must be informed that any data output from the data cache will be invalid so that any pending coprocessor load instructions must be flushed. The coprocessor or system register block must take appropriate action in order to ensure the correct semantic on the TCB bus.</td></tr>
<tr><td>tcb_cpustalled</td><td>Out</td><td colspan="4">Asserted by the CPU if the CPU LS pipeline was stalled in the previous cycle. The coprocessor or system register block must take appropriate action in order to ensure the correct semantic on the TCB bus.</td></tr>
<tr><td>tcb_wdataa[31:0],
tcb_wdatab[31:0]</td><td>Out</td><td colspan="4">Data to write to coprocessor. For word transfers, tcb_wdataa contains the data to transfer, and tcb_wdatab is UNDEFINED. For doubleword transfers, tcb_wdataa contains the most significant part of the data, and tcb_wdatab contains the least significant part of the data.</td></tr>
</table>

www.BDTIC.com/ATMEL

Table 8-1. TCB signals

Name	Dir	Description
tcb_ready	In	Acknowledge signal from the coprocessor slaves. Indicates if the coprocessor or system register was able to process the command from the master. If not, the LS pipeline should stall until the tcb_ready signal is asserted. Each slave connected to the TCB has an individual ready signal, and all these are AND-ed together to form tcb_ready. All slaves connected to the TCB should therefore leave their ready signal HIGH AT ALL TIMES unless they really want to stall the TCB. If one or more slaves drive their ready signal low, the TCB and LS pipe is stalled. A TCB slave may only start a stall during the address phase of the TCB operation. If tcb_ready stays asserted on the first clock edge after the operation is initiated, tcb_ready must stay asserted until the data is ready.
tcb_present[7:0]	In	Indicates which coprocessors are present on the TCB bus. AVR32 supports 8 coprocessors, and each of the 8 coprocessor addresses is either in use by a coprocessor or not in use. When attaching a coprocessor on the TCB bus, the system integrator must assert the corresponding bit position in the tcb_present bus to 1. The bit position of unconnected TCB addresses must be disasserted. The tcb_present bus is used by the ID stage when decoding a coprocessor instruction, in order to detect whether a coprocessor absent exception should be triggered. The tcb_present signal is static, ie. it should not change during execution. If bit n asserted, coprocessor n is present. Otherwise, no coprocessor with address n is present on the TCB bus.
tcb_rdataa[31:0] tcb_rdatab[31:0]	In	Data read from the coprocessor to the CPU. The supported data widths are word and doubleword. Each slave connected to the TCB has individual tcb_rdataa and tcb_rdatab outputs, and all these are AND-ed together to form the tcb_rdataa and tcb_rdatab that is input to the CPU. All slaves connected to the TCB should therefore leave their tcb_rdataa and tcb_rdatab outputs HIGH AT ALL TIMES unless they really want to perform a write to the TCB. For word transfers, tcb_rdataa contains the data to transfer, and tcb_rdatab is UNDEFINED. For doubleword transfers, tcb_rdataa contains the most significant part of the data, and tcb_rdatab contains the least significant part of the data.

8.3 Connecting coprocessors to the TCB bus

Connecting new coprocessors to the TCB bus is simple. Just assert the bit number in tcb_present corresponding to the desired coprocessor number, and AND the tcb_ready, tcb_rdataa and tcb_rdatab signals from the coprocessor together with the same signals from the other coprocessors. The TCB signals output from the CPU must also be routed to the corresponding inputs on the coprocessor.

8.4 Execution of coprocessor instructions

All coprocessor instructions flow through the LS pipeline. The state machine in the DA stage is responsible for correct execution of the coprocessor instructions. The coprocessor data transfer instructions behave very similarly to their corresponding load/store instructions. The main difference is that data is written/read from the TCB bus instead of the integer register file. The timing

www.BDTIC.com/ATMEL

requirements of the TCB bus must be obeyed. This may require stalling of the LS pipeline for the required number of clock cycles.

8.4.1 Stalling of the TCB bus and the LS pipeline

An addressed TCB slave may stall the TCB bus and the LS pipeline if it is unable to fulfill the required TCB timing. The addressed slave can perform this stalling by outputting a logical-0 value on its tcb_ready output. Writing a value of zero on tcb_ready will cause the LS pipeline and the data cache to stall until tcb_ready is written to one again. When the CPU LS pipeline stalls, all outputs from the CPU to the TCB bus will remain unchanged.

If the CPU LS pipeline stalls for some reason, the TCB may need to be informed if the LS stall affects a TCB transfer. The tcb_cpustalled signal is a registered version of the stall signal in the LS pipeline. If the CPU was stalled in the previous cycle, tcb_cpustalled is asserted. Otherwise, tcb_cpustalled is disasserted. Since disassertion of tcb_ready will cause the LS pipeline to stall, tcb_cpustalled will always be high the cycle following a cycle where tcb_ready was low.

8.4.2 Coprocessor operation

The *cop* instruction issues a command in the form of an opcode and three operand addresses to the addressed coprocessor. The coprocessor operation only uses the address phase of the bus transaction, while the data phase is unused and all signals are considered don't care.

If the CPU stalls immediately after issuing a coprocessor operation, a coprocessor operation opcode will be present on tcb_cmd for multiple cycles. If the TCB slave needs to avoid the operation being executed several times, the TCB slave must qualify any coprocessor operation opcodes with tcb_cpustalled being low.

8.4.3 Writing data to coprocessor

There are several instructions that transfer data to coprocessors. These instructions are: *ldc.d*, *ldc.w*, *ldcm.d*, *ldcm.w*, *mvrc*, *mtdr* and *mtsr*. *Mtdr* and *mtsr* are not coprocessor instructions, but use the TCB in a manner very similar to *mvrc*, and is therefore included here. Data can be transferred to coprocessor registers in sizes of word and doubleword. The data transferred to the coprocessor register file is either read from memory or from one of the integer registers in the CPU.

The *ldcm* instructions behave similarly to the *ldm* and *popm* instructions. The hardware always try to transfer a doubleword from the cache in order to speed up the data transfer. This is successful if the memory pointer is doubleword aligned. Otherwise, a word access is performed first, then the remaining transfers are performed as doubleword accesses.

The FSM in the DA stage decomposes load of multiple registers into a sequence of load of words and doublewords. These accesses are pipelined according to the TCB rules, allowing a bus transfer of word or doubleword per clock cycle.

If the CPU stalls after issuing a TCB write command, tcb_wdataa and tcb_wdatab will contain the write data for the previous TCB bus cycle. This write data will be present on the bus until the last cycle where tcb_cpustalled is high. In this cycle, the write data for the stalled TCB write command will be present on the bus. If the TCB slave needs to avoid that the write data for the previous TCB command is written to the TCB slave destination registers for the stalled (current) write, the TCB slave must not update the TCB destination registers when tcb_cpustalled is high.

www.BDTIC.com/ATMEL

8.4.4 Reading data from coprocessor

There are several instructions that transfer data from coprocessors. These instructions are: *stc.d*, *stc.w*, *stcm.d*, *stcm.w*, *mvcr, mfdr* and *mfsr*. *Mfdr* and *mfsr* are not coprocessor instructions, but use the TCB in a manner very similar to *mvcr*, and is therefore included here. Data can be transferred from coprocessor registers in sizes of word and doubleword. The data transferred from the coprocessor register file is either stored to memory or into one of the integer registers in the CPU.

The *stcm* instructions behave similarly to the *stm* and *pushm* instructions. The hardware always try to transfer a doubleword to the cache in order to speed up the data transfer. This is successful if the memory pointer is doubleword aligned. Otherwise, a word access is performed first, then the remaining transfers are performed as doubleword accesses.

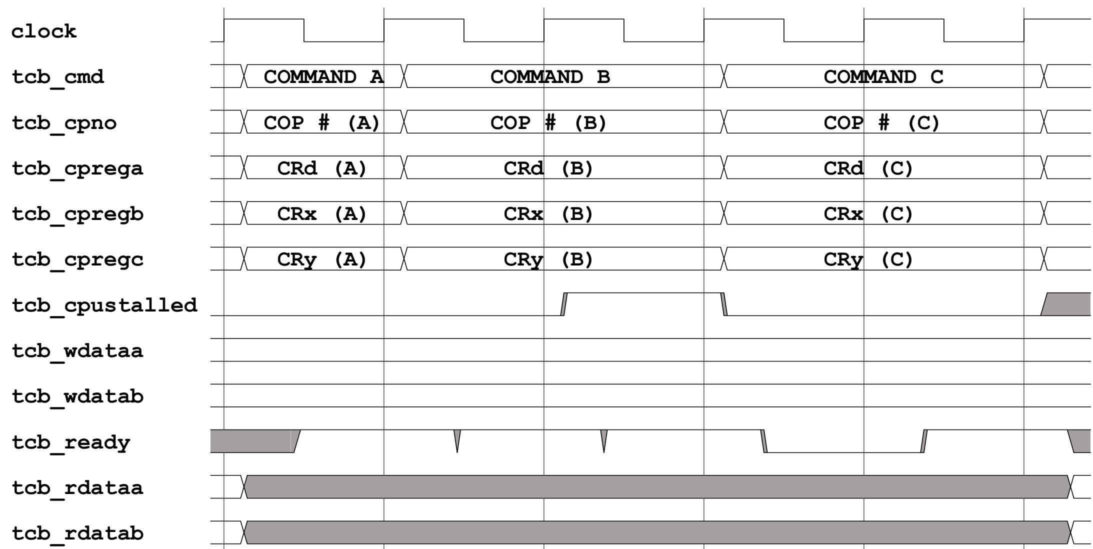
The FSM in the DA stage decomposes store of multiple registers into a sequence of store of words and doublewords. These accesses are pipelined according to the TCB rules, allowing a bus transfer of word or doubleword per clock cycle.

A TCB slave must take special action if it was read from in the previous cycle, and tcb_cpustalled gets asserted. In this case, the slave must continue to output the data values it put on tcb_rdataa and tcb_rdatab in the previous cycle for as long as tcb_cpustalled is asserted, even though a new command is present on tcb_cmd.

8.5 Timing diagrams

8.5.1 Coprocessor operation

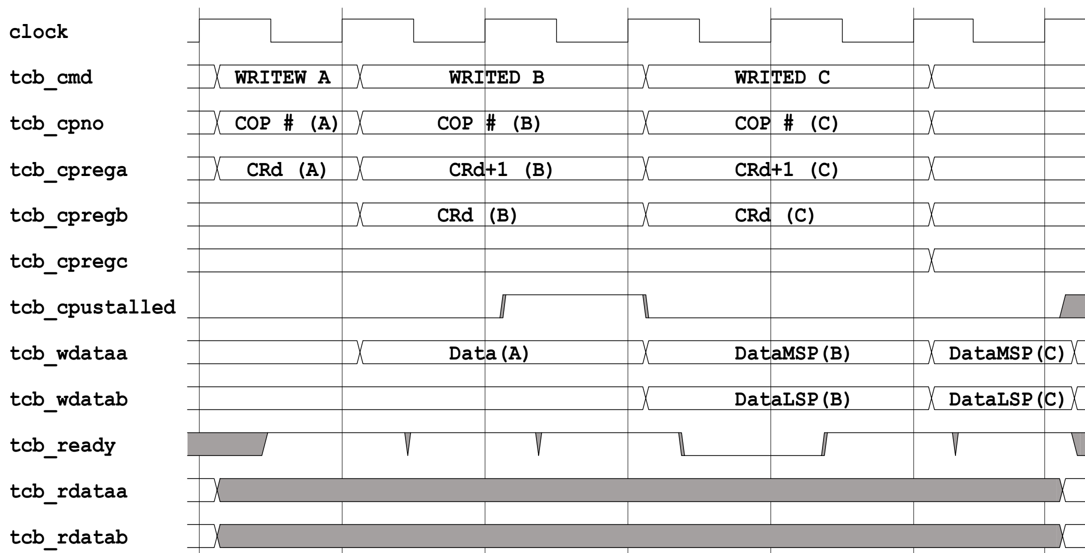
Figure 8-2. COP bus timing.



www.BDTIC.com/ATMEL

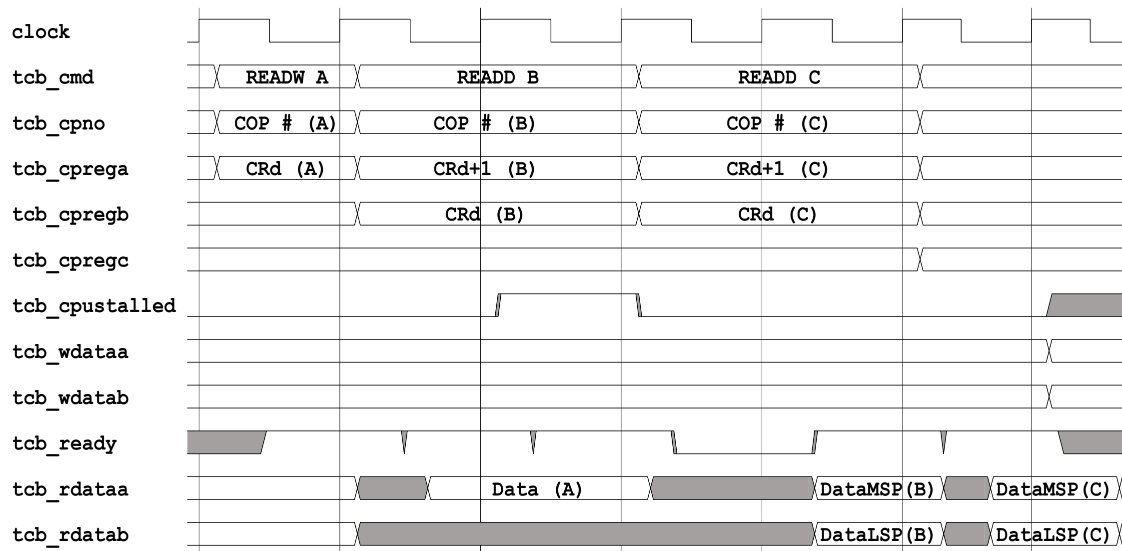
8.5.2 Writes to coprocessor register file

Figure 8-3. Write to CP timing.



8.5.2.1 Reads from coprocessor register file

Figure 8-4. Read from CP timing.



www.BDTIC.com/ATMEL

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 AP 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 read/write access to data memory and data cache
- Real-time process trace
- ASID-specific breakpoints

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.

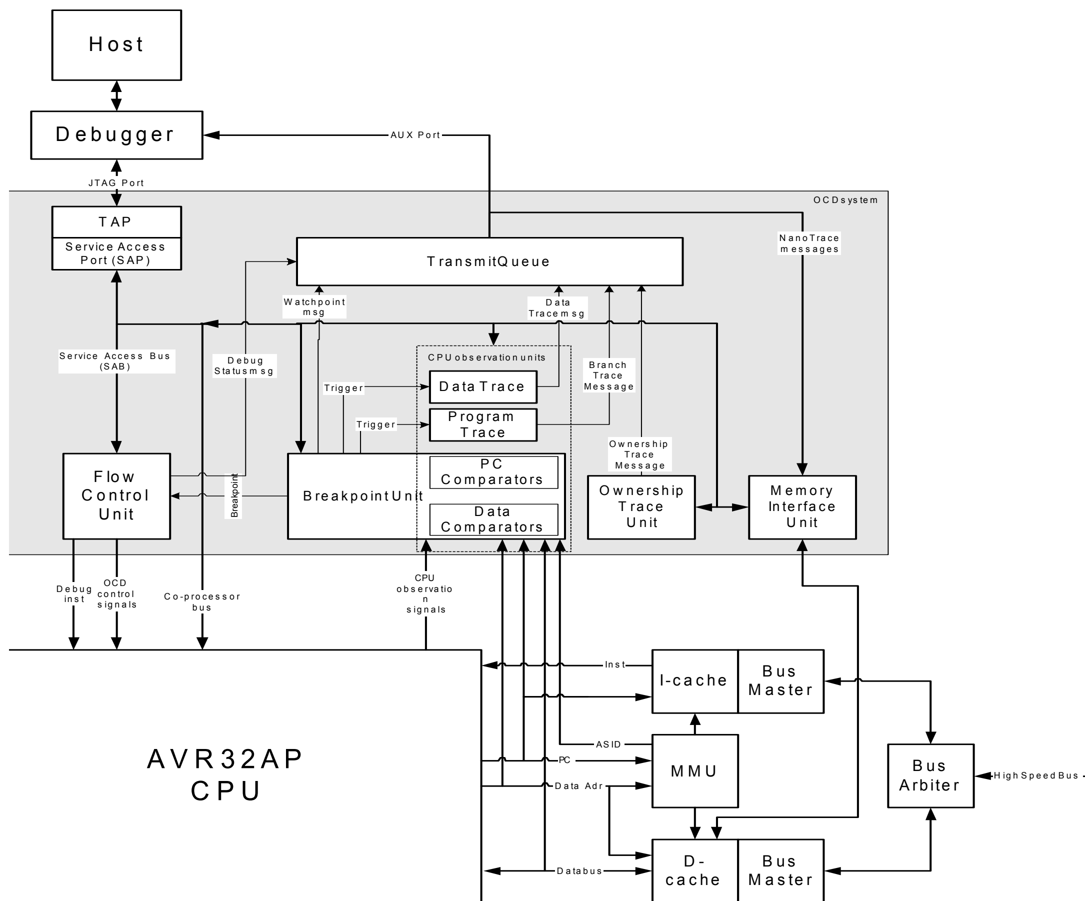
9.1.2.1 Host, debugger, and emulator

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 mem-

www.BDTIC.com/ATMEL

ory 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.

www.BDTIC.com/ATMEL

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 RTOS debug support

Applications developed on an RTOS 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

www.BDTIC.com/ATMEL

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.

9.1.2.9 Real-time memory access

Real-time block transfers of data to or from system memory is also possible through the Memory Interface Unit (MIU). The tool initiates these transfers by writing to OCD registers in the MIU. Unlike the comparator units, the MIU operates on physical addresses, since no interference with the operating system can be expected. This means that the debug software must perform the translation between the virtual and physical address map before accessing the memory. This mapping is typically specified through page tables located in a privileged, unmapped area of the RAM, and can be read out by the debugger to calculate the physical address. Since the location and format of the page table is OS specific, the debugger must be "OS aware" to employ this feature.

The CPU can also use the MIU to perform an efficient transfer of data from user memory to the tool, without a prior read request from the tool.

9.1.2.10 Java debug features

AVR32 AP has native support for Java bytecode programs, executing on a Java Virtual Machine (JVM) platform. The OCD features mentioned above are also available in Java mode, enabling the same debug support for Java programs as for C/C++/assembly programs. The JVM will implement a debug protocol which the debugger can use to extract key information about tasks and objects in the execution environment. Alternatively, if the format of the data structures created by the JVM is known by the debugger, the debugger can read out all JVM and task information by block read commands.

www.BDTIC.com/ATMEL

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 3 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 Return Address 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.

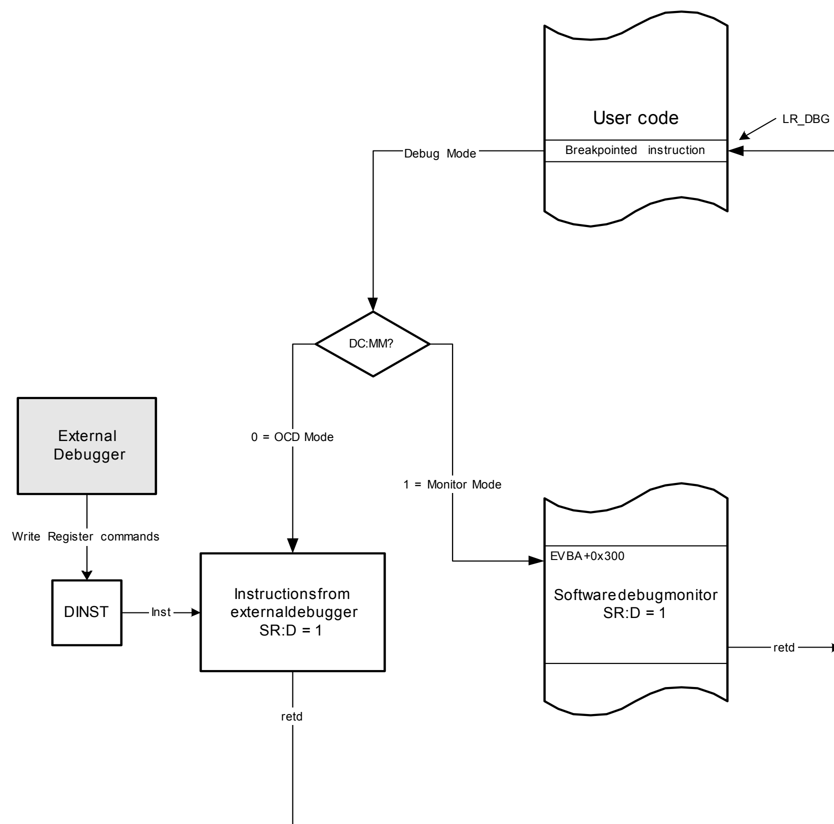
Debug Mode is exited by the *retd* instruction, both in Monitor Mode and OCD Mode. This restores PC from RAR_DBG and SR from RSR_DBG.

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).

www.BDTIC.com/ATMEL

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.

www.BDTIC.com/ATMEL

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.

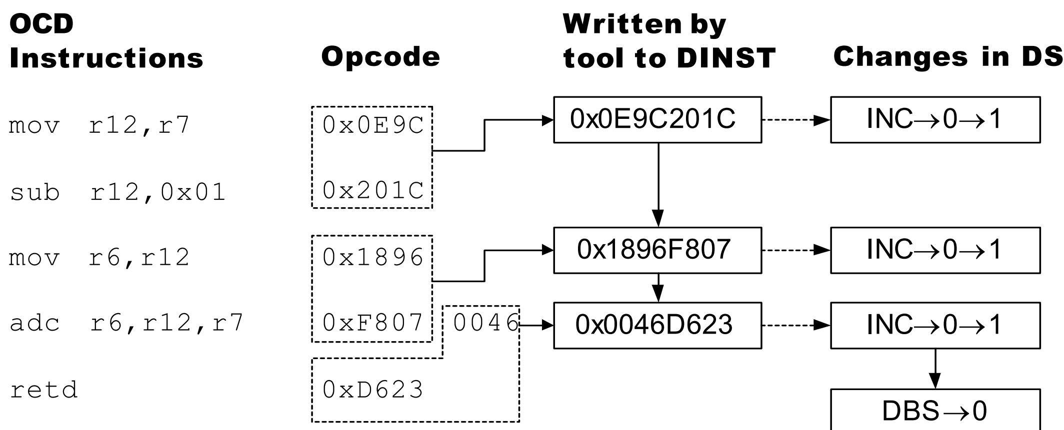
All OCD units can be configured while the CPU executes in OCD Mode, but the following debug features are disabled:

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

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

www.BDTIC.com/ATMEL

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
- Hardware error
- Event on $\overline{\text{EVTI}}$ pin
- NanoTrace buffer full
- 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 break-pointed instruction is executed.

www.BDTIC.com/ATMEL

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.

Alternatively, the Instruction Cache memory can be directly written by the debugger through the JTAG port. The page containing the software breakpoint can be programmed into a cache page and locked, to prevent it from being flushed. Every time the CPU executes the breakpointed section of the code, it fetches these instructions from the cache instead of the program memory. This method can only be used to insert software breakpoints in cacheable regions of the memory space, as defined by the Memory Management Unit.

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

www.BDTIC.com/ATMEL

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.

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 Hardware Error

The CPU might encounter problems which cannot be handled in software. This includes accessing a memory area reserved for NanoTrace. These types of errors should never occur in a correctly written application, and will normally trigger a soft reset.

To ease debugging of these types of errors, the debugger can write the DC:TSR (Trap Soft Reset) bit to one. The CPU will then enter Debug Mode if a soft reset occurs. This includes any kind of soft reset in the device, such as watchdog reset. The Hardware Error bit (HWE) in the Development Status register will be set to indicate that a trapped soft reset caused entry to OCD mode.

Note that if OCD mode is disabled (i.a. also when Monitor Mode is enabled), the soft reset allows the software to restart in a defined manner.

Since the soft reset causes may corrupt CPU execution, the RAR_DBG and RSR_DBG are undefined when Debug Mode is entered due to a hardware error.

9.2.4.8 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.

9.2.4.9 NanoTrace buffer full

When using NanoTrace to write trace information to memory, the user can configure a breakpoint when the buffer becomes full. This will set the NanoTrace Buffer Full (NTBF) bit in DS. RAR_DBG will point to the last non-executed instruction.

www.BDTIC.com/ATMEL

9.2.4.10 *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).

www.BDTIC.com/ATMEL

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.

www.BDTIC.com/ATMEL

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:

1. Allow the CPU to wake up from sleep mode on a debug request.
2. 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
NanoTrace	No
Direct Memory Access	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 152. 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.

www.BDTIC.com/ATMEL

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
Read/Write Access Control/Status (RWCS)	Yes	No
Read/Write Access Address (RWA)	Yes	No
Read/Write Access Data (RWD)	Yes	No
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 Optimization Control (DOC)	No	No

www.BDTIC.com/ATMEL

Table 9-3. OCD Register access (Continued)

Register	Can be written by JTAG while CPU is running?	Can be written by CPU in Monitor Mode?
Event Pair Control (EPC)	Can only be written while breakpoint / watchpoint is disabled	Yes, if SR:DM is set or breakpoint disabled.
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 Debugging Java programs

9.2.12.1 Java mode operation

To run Java programs, a Java Virtual Machine (JVM) must be implemented in software. Java bytecode programs can then be executed natively on the CPU by placing the CPU in Java mode. This mode is described in the "AVR32 CPU Architecture" document. The Java mode characteristics include:

- The CPU decodes instructions as Java bytecodes, each consisting of 1 or more bytes.
- Complex Java instructions are trapped and executed as a RISC routine embedded in the JVM.
- Java programs can execute in Application or Supervisor mode, thus using the Application or Supervisor register context, respectively.
- The lower half of the register file is remapped to operate as a push/pop stack for operands
- Other register file registers and system registers hold pointers to memory structures created by the JVM.

9.2.12.2 Java and debug functionality

The operating mode of the CPU is contained in the bits in the upper half of the Status Register (SR) in the CPU. When Debug Mode (OCD or Monitor Mode) is entered from Java mode, the CPU switches to RISC mode (SR:D=1, SR:J=0) and the exception register context (SR:M=6). By changing SR:M to Application or Supervisor mode, the debugger can execute RISC instructions on the CPU and still read out the register context of the Java program. The SR:J bit should never be set in Debug Mode, as this can cause undefined behavior of the CPU.

The debug features available in RISC mode are also available for Java mode. Note the following particularities about Java debugging:

- Software breakpoints are set by the Java breakpoint bytecode instead of the RISC breakpoint opcode.
- Instruction replacement is possible when exiting from Debug Mode to Java mode. The same procedure as for RISC mode can be used. The Java bytecode for the return instruction must be written to the Debug Instruction Register, and the address of this instruction must be written to the Debug Program Counter.
- If the memory allocation scheme for the JVM implementation is known, memory block read commands can be used to extract any task information created by the JVM.

www.BDTIC.com/ATMEL

- The incjosp RISC instruction is only used in JVM implementations. This instructioncannot be breakpointed. Single stepping over the instruction will result in stepping over the next instruction as well.

9.2.13 CPU optimization control

The CPU contains a number of optimization features which could obscure visibility and complicate debugging. These features are normally controlled by the CPUCR system register, but are automatically disabled by the OCD system according to which OCD features are enabled.

It is possible to for the debugger to override the default disabling of CPU optimization features by writing the CPU Control Mask Register. The debugger can thus manually tune which features should be disabled to enhance debugging of performance critical code, trading CPU performance against debug visibility. CPUCM will retain its written value until written with a new value or the OCD is reset.

The user should be familiar with the operation of the CPU pipeline and Data Cache to utilize these optimization features. Changing the CPUCM register is only recommended for advanced users who require high CPU performance during their debug sessions.

9.2.13.1 Branch prediction

By default, the CPU will optimize branch execution by attempting to predict the target address for the branch. This has an adverse effect for program trace, since extended branch (br{cond4}) and extended rcall (rcall k21) could generate messages with incorrect target addresses.

The OCD system automatically disables this feature when program trace is enabled. If the code does not contain extended rbanch or rcall instructions, or the user accepts incorrect target addresses for these instructions, the user can write CPUCM:BEM to one.

It is not recommended to write both CPUCM:BEM and CPUCM:FEM to one during program trace, as this will cause many errors in the program trace output.

9.2.13.2 Branch folding

The CPU pipeline can compress a branch instruction and the instruction following the branch into one pipeline instruction, to improve instruction throughput. This process is known as branch folding. If branch folding is enabled, the OCD is unable to observe the PC of a folded branch, which can cause PC breakpoints on branch instructions to failt to trigger. Branch folding is therefore by default disabled when the OCD is enabled.

Branch folding can be kept enabled by writing CPUCM:FEM to one. This means that instructions following a branch can not always be breakpointed.

9.2.13.3 Return stack

To speed up subroutine and interrupt handling, the CPU can buffer the most recently used return addresses in the Return Stack instead of having to fetch them from the regular data memory stack.

If returning from a subroutine using a ld/ldm or popm to PC, and this address is fetched from the return stack, program trace will not report an Indirect Branch message, as it should.

For this reason, the OCD system disables the return stack when program trace is activated. If the code does not contain loads or pop to PC, the user can keep the return stack enabled by writing CPUCM:REM to one.

www.BDTIC.com/ATMEL

9.2.13.4 *Imprecise breakpoints*

The CPU will normally issue more instructions for execution before previous instructions are completed. When breakpointing memory operations which cause exceptions, the exception may already have been started. This causes the breakpoint to behave incorrectly, typically triggering for a later instruction instead. To avoid confusion, the OCD ensures breakpoints are precise when the OCD is enabled. This forces the CPU to delay the exception check for memory operations until a possible breakpoint has been resolved. This normally causes one cycle penalty for memory operations.

It is possible, but not recommended, to allow imprecise breakpoints during debugging by writing CPUCM:IBEM to one.

9.2.13.5 *Imprecise execution*

The AVR32 AP CPU contains logic to optimize instruction execution, which implies that instructions may complete out-of-order. In a debug context, this may lead to imprecise behavior. Specifically, when a data breakpoint triggers, one or more instructions following the breakpoint may have been executed. Also, when using DC:OVC to prevent data trace overruns, several memory operations may have already been started, possibly causing an overrun situation.

For this reason, the OCD system forces the CPU to use precise execution when data breakpoints are enabled or DC:OVC prevents data trace overruns. Precise execution will reduce memory performance significantly. Imprecise execution can be kept enabled by writing CPUCM:IEEM to one.

9.2.14 Messages

9.2.14.1 *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.
- 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

www.BDTIC.com/ATMEL

9.2.15 Registers

9.2.15.1 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

9.2.15.2 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

www.BDTIC.com/ATMEL

Table 9-6. Nexus Configuration Register

R/W	Bit Number	Field Name	Init. Val.	Description
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 = AVR32 AP OCD Other = Reserved
R	3:0	NXARCH	0	Architecture 0000 = AVR32B 0001 = AVR32A Other = reserved

9.2.15.3 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

9.2.15.4 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

www.BDTIC.com/ATMEL

9.2.15.5 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.

Table 9-9. Debug Communication Status 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	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.

9.2.15.6 Development Control Register (DC)

DC is used for basic development control of the CPU.

Table 9-10. 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

www.BDTIC.com/ATMEL

Table 9-10. Development Control Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	27	RID	0	RID - Run In Debug 0: Peripherals are frozen in Debug Mode 1: Peripherals keep running in Debug Mode
R/W	26	TSR	0	TSR - Trap Soft Reset 0: A soft reset event causes the CPU to be reset 1: A soft reset event causes the CPU to enter Debug Mode.
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.
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.

www.BDTIC.com/ATMEL

Table 9-10. Development Control Register

R/W	Bit Number	Field Name	Init. Val.	Description
	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.
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.

9.2.15.7 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.

www.BDTIC.com/ATMEL

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

Table 9-11. 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 was entered due to the NanoTrace buffer being 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.

www.BDTIC.com/ATMEL

Table 9-11. 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	HWE	0	HWE - Hardware Error This bit is set if a hardware error has triggered entry to Debug Mode. The debugger should assume that all status information has been lost, and write the RES bit in DC to reset the system. The OCD control and configuration registers should be reconfigured.
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.

9.2.15.8 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 9-12. 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.

9.2.15.9 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.

ATMEL®
www.BDTIC.com/ATMEL

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 9-13. 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

9.2.15.10 CPU Control Mask Register (CPUCM)

This register prevents the OCD from overriding the operation of the CPU Control Register (CPUCR). A value written to this register is kept until a new value is written or the OCD is reset..

Table 9-14. CPU Control Mask Register

R/W	Bit Number	Field Name	Init. Val.	Description
R	31:6	Reserved	0	
R/W	5	IEEM	0	Imprecise Execution Enable Mask When set, the OCD will not disable imprecise execution.
R/W	4	IBEM	0	Imprecise Breakpoint Enable Mask When set, the OCD will not disable imprecise PC breakpoints.
R/W	3	REM	0	Return stack Enable Mask When set, the OCD will not disable the return stack.
R/W	2	FEM	0	Branch Folding Enable Mask When set, the OCD will not disable branch folding.
R/W	1	BEM	0	Branch Prediction Enable Mask When set, the OCD will not disable branch prediction.
R/W	0	Reserved	0	

www.BDTIC.com/ATMEL

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 9-15. The signals are described in Table 9-16.

Table 9-15. 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 9-16. JTAG signals

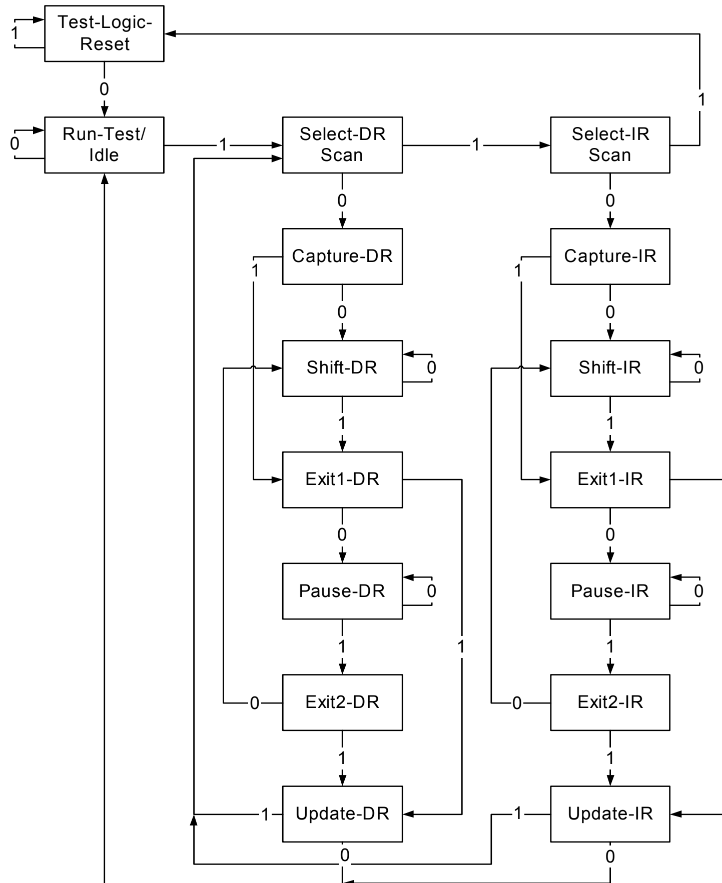
Pin	Direction	Description
TRST_N	Input	Asynchronous reset for the TAP controller and JTAG registers
TCK	Input	Test Clock. Data is driven on falling edge, sampled on rising edge.
TMS	Input	Test Mode Select
TDI	Input	Test Data In

www.BDTIC.com/ATMEL

Table 9-16. JTAG signals

Pin	Direction	Description
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 AVR32 AP.

- 6 data output pins (MDO)
- 2 message start/end output pins (MSEO)
- 1 EVTO pin
- 1 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.

www.BDTIC.com/ATMEL

The AUX pins may be multiplexed with GPIO in a device. By default, the MCKO, MDO, and MSEO pins are tristated or used as GPIO, and the Nexus functionality must be explicitly enabled by the debugger. EVTO, 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 9-17.

Table 9-17. 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 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 (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 MSEO shall occur on rising edge of MCKO. In double datarate mode, external latching of MSEO shall occur on both edges of MCKO.
EVTO	1	O	Event Out (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 (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

www.BDTIC.com/ATMEL

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 9-18.

Table 9-18. 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*

Message transmission can be enabled or disabled according to the state of the $\overline{\text{EVTI}}$ pin when the JTAG TAP controller is reset. When messaging is enabled, output messages are transmitted normally. If message transmission is disabled, the auxiliary output pins (MCKO, MDO, $\overline{\text{MSEO}}$) are tristated, and no messages will be transmitted.

Reset configuration information must be valid on $\overline{\text{EVTI}}$ at least 2 TCK periods prior to negation of $\overline{\text{TRST}}$ or exit from the TEST-LOGIC-RESET TAP state. The AUX port will be enabled as shown in Table 9-19.

If the Nexus port is disabled after reset, the debugger can still enable the port by writing to the AXC:AXE (Auxiliary Enable) bit to enable trace functionality at any time before trace is activated.

www.BDTIC.com/ATMEL

Debug functionality based on the JTAG or $\overline{\text{EVTI}}$ or $\overline{\text{EVTO}}$ pins is still available even if the AUX port is disabled.

Table 9-19. $\overline{\text{EVTI}}$ pin reset configuration

Reset state	Description
0	Message transmission enabled
1	Message transmission disabled (default)

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, $f_{AUX} = f_{CPU}/2$.

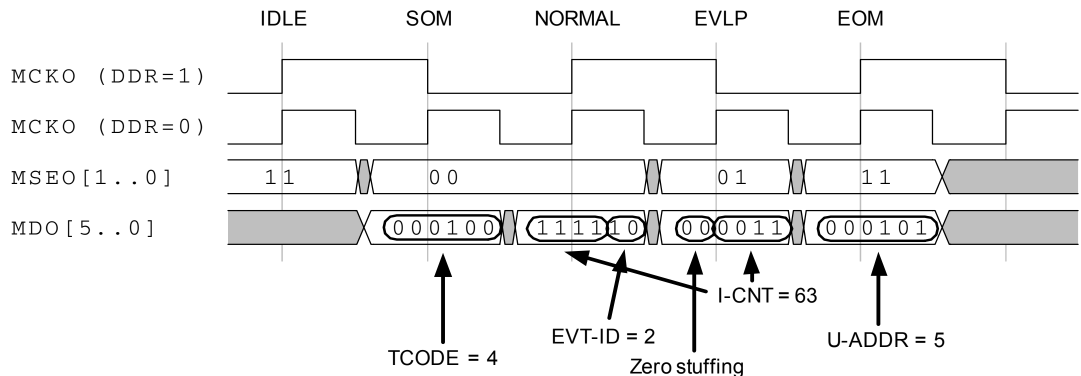
www.BDTIC.com/ATMEL

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.

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.

www.BDTIC.com/ATMEL

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

www.BDTIC.com/ATMEL

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	31:14	Reserved	0	Reserved These bits are reserved, and will always read as 0
R/W	13	REXTEN	0	This bit is reserved for internal test purposes and should be written to zero.
R/W	12	REX	0	This bit is reserved for internal test purposes and should be written to zero.
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	AXS	0	AXS - Auxiliary Port Select 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/W	8	AXE	0	AXE - Auxiliary Port Enable 0: AUX port is tristated 1: AUX port is used for Nexus operation.
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.

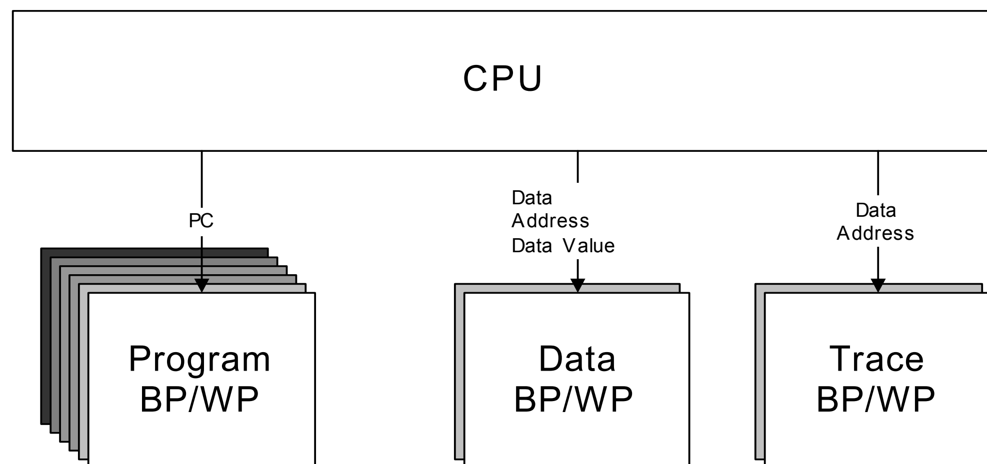
www.BDTIC.com/ATMEL

9.4 Breakpoints

9.4.1 Overview

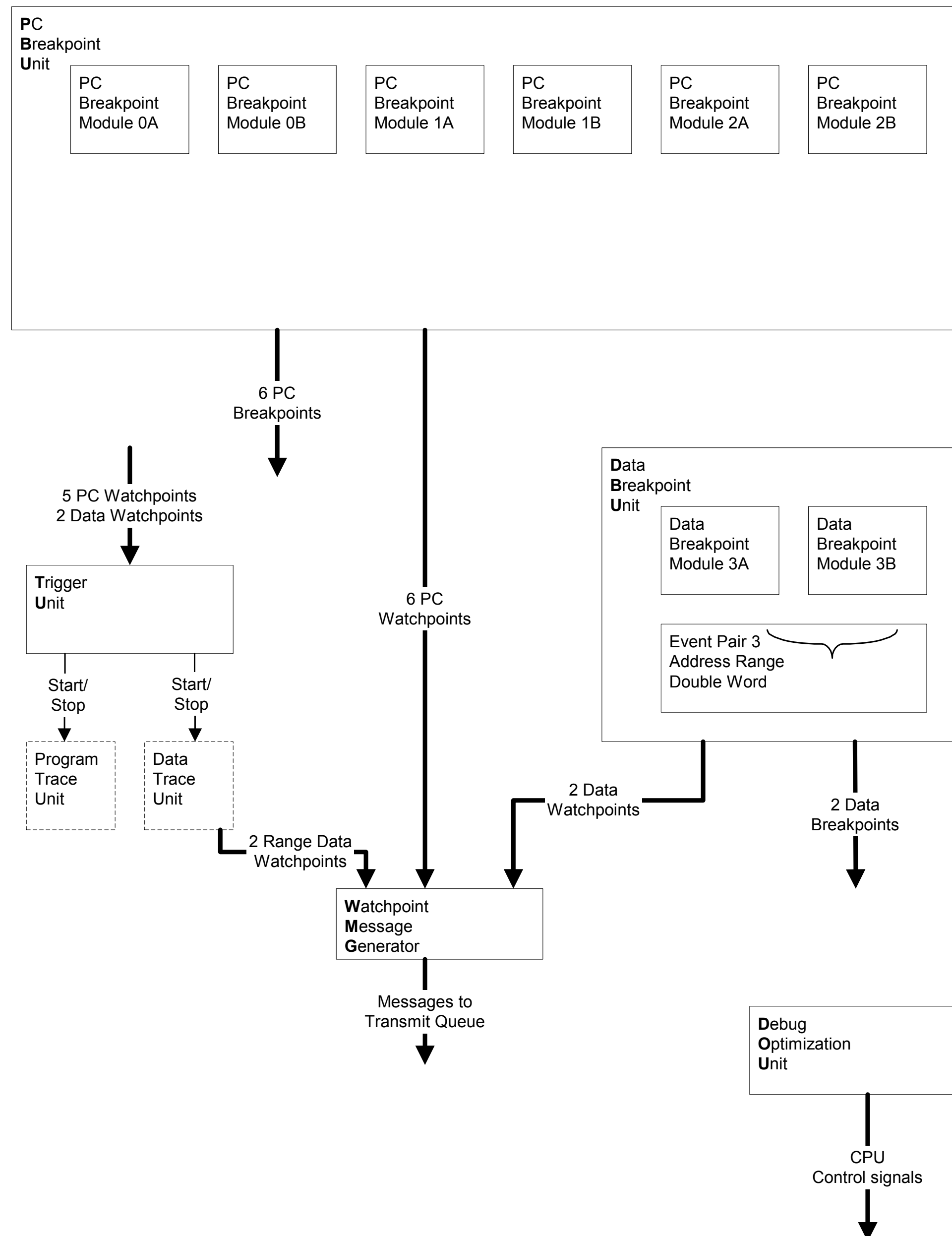
The Nexus Recommended Register map supports up to 8 universal breakpoints. However since the AVR32 AP hardware employs separate instruction and data memories, the OCD system must also separate program and data breakpoints. Any breakpoint can also be programmed as a watchpoint. The watchpoint will trigger 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.



www.BDTIC.com/ATMEL

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, as well as EPC3.

www.BDTIC.com/ATMEL

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.

BWA operates on virtual addresses. In order to get a precise match on a virtual address if MMU is enabled, Address Space Identifier (ASID) matching must be enabled in the BWC, and the ASID must be written to the ASID field of the BWC. If the ASID to match on will be read from the current ASID in the MMU.

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 Return Address 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.

The AME bit in the BWCA registers can be used to enable a bitwise address masking. When AME is enabled the BWA register in the B module is used as a noninverting bitwise mask that is applied to the PC and the value in BWA register in the A module. The A breakpoint will thus trigger when PC = BWAnA & BWAnB. The B breakpoint will never trigger when AME is enabled.

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.

www.BDTIC.com/ATMEL

9.4.2.3 Data Breakpoints

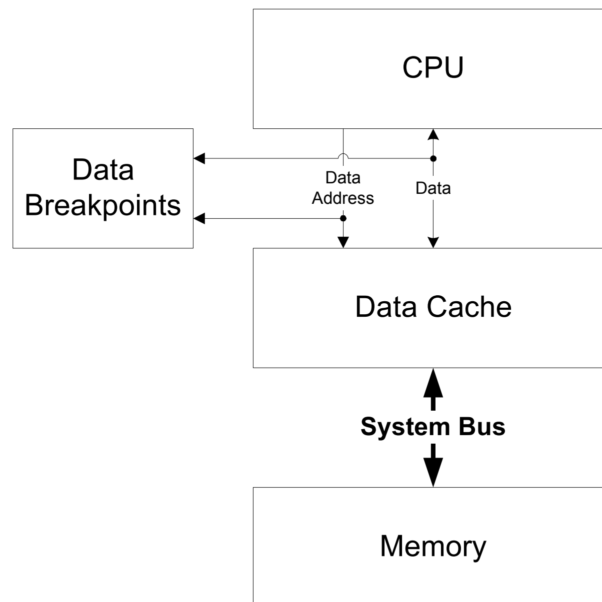
Data Breakpoint modules listen on the data address and data value lines between the CPU and the data cache and can halt the CPU, or send a watchpoint message, if the address and / or value meets a stored compare value. Unlike program breakpoints, data breakpoints halt on the next instruction after the 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. BWA matches on virtual addresses. In order to get a precise match on a virtual address when MMU is enabled, the Address Space Identifier (ASID) matching must be enabled in the BWC, and the ASID must be written to the ASID field of the BWC. The ASID to match on will be read from the current ASID in the MMU.

As shown in Figure 9-8 the data breakpoint modules snoop on the address and data lines between the CPU and the data cache. This ensures that the data breakpoints only trigger on actual load / store operations and not on prefetch or other automated cache related accesses. This is required for data breakpoints to be consistent with the CPUs view of the data memory. It does however, have the effect that a write to cached memory will trigger before the write has been flushed to memory. Uncached writes will trigger as they are written to the memory.

Data breakpoints are not available in Monitor Mode.

Figure 9-8. Data breakpint interface.



9.4.2.4 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 9-9. 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.

www.BDTIC.com/ATMEL

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 9-9, 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).

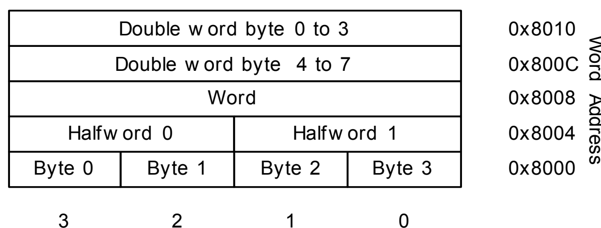
By default, the data breakpoint module will match on the data value regardless of the size of the access. The data BWC can also be set to match on a specific access size if the SIZE bits are set. The debugger can for example, set the breakpoint module to match only on byte writes to byte 1 in Figure 9-9. 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.

The OCD system can also break on a 64-bit value if both data breakpoint modules are combined as shown in Figure 9-10. Setting the Double Word Enable (DWE) bit in EPC3 will cause breakpoint module A to trigger only if both breakpoint module A and B matches. The debugger can then set the BWA3A and BWD3A to the least significant word in the double word and the BWA3B and BWD3B to the most significant word of the double word. BWC3A:BWE control the breakpoint operation of the combined breakpoint, while BWC3B:BWE is disregarded.

For example, to set a breakpoint when the address 0x800C is written to 0x0123456789ABCDEF, the registers must be configured as follows:

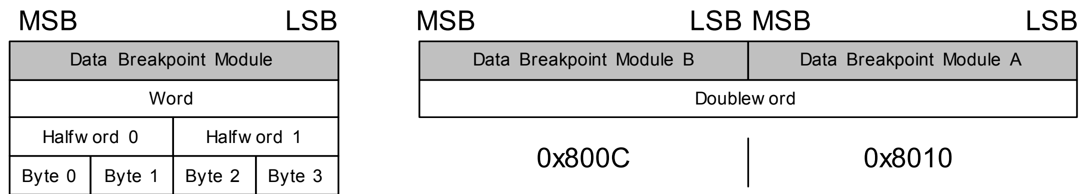
- EPC3 = (DWE)
- BWC3A = (BWE | BRW | BWO*3 | SIZE*7)
- BWC3B = (BWE | BRW | BWO*3 | SIZE*7)
- BWA3A = 0x8010
- BWA3B = 0x800C
- BWD3A = 0x89ABCDEF
- BWD3B = 0x01234567

Figure 9-9. Memory access data alignment.



www.BDTIC.com/ATMEL

Figure 9-10. Data breakpoint alignment.



9.4.2.5 *Unaligned accesses*

The CPU supports unaligned accesses by breaking the operation down into multiple aligned accesses. This means that one ld/st instruction from the CPU can be seen as many sequential operations on the databus, depending on the alignment of the data to be accessed:

- Unaligned double word load / store is seen as a sequence of word loads / stores.
- Unaligned store word is seen as a sequence of stores which may have different sizes. Eg st.b , st.h, st.b for byte aligned st.w
- Unaligned load word is always done as two load words.

9.4.3 Advanced features

9.4.3.1 *Ranges*

It is possible to compine both data breakpoint modules to break on a range of data addresses. Range is enabled using the EPC3 register. Whenever a data breakpoint range is used data value matching should be disabled.

The debugger can set up an event pair to give a breakpoint or watchpoint on a range of instruction addresses. The event pair will then configure the A and B address comparator to give a match if the instruction address is less than or equal to the BWA. Using the Range (RNG) bits of the Event Pair Control (EPC) register either inclusive or exclusive ranges can then be set up as shown in Table 9-23.

When ranges are enabled, the BWC:BWE of module A will control the ranged breakpoint, module B will be disabled.

Table 9-23. Range settings.

EPC3: RNG	Resulting Range
10	BWA3A < ADDR <= BWA3B
01	ADDR <= BWA3A or ADDR > BWA3B

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.

www.BDTIC.com/ATMEL

9.4.5 Messages

9.4.5.1 Watchpoint Hit (WH)

Table 9-24. 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 9-25. 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 9-26. PC BWAnx Register

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

www.BDTIC.com/ATMEL

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

Table 9-27. 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
R	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	13:9	Reserved	0	Reserved
RW	8:1	ASID	0x00	ASID - Asid to match The 8 bit ASID to match when ASID matching is enabled.
RW	0	ASIDEN	0	ASIDEN - ASID match enable 0 = Disabled. 1 = Enabled. The breakpoint module will only give a match if the ASID also matches.

www.BDTIC.com/ATMEL

9.4.6.3 *Event Pair Control 3 (EPC3)*

Table 9-28. Data Event Pair Control (EPC3) Register

R/W	Bit Number	Field Name	Init. Val.	Description
R	31:3	-		Reserved
RW	2	DWE	0	DWE - Enable combined Double Word value compare 0 = Disabled (Default) 1= Combine two event units to do 64 bit compare If enabled both data breakpoint modules units will be combined to do a single 64 bit value compare
RW	1:0	RNG	0b00	RNG - Range Enable 00: disabled 01: Exclusive range (PC <= Even or PC > Odd) 10: Inclusive range (Even < PC <= Odd) 11: Reserved

9.4.6.4 *Data Breakpoint / Watchpoint Address (BWA3A, BWA3B)*

Table 9-29. 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.

9.4.6.5 *Data Breakpoint / Watchpoint Data (BWD3A, BWD3B)*

Table 9-30. 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.

www.BDTIC.com/ATMEL

9.4.6.6 Data Breakpoint / Watchpoint Control (BWC3A, BWC3B)

Table 9-31. 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	15:12	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
RW	8:1	ASID	0x00	ASID - Asid to match The 8 bit ASID to match when ASID matching is enabled.
RW	0	ASIDEN	0	ASIDEN - ASID match enable 0 = Disabled. 1 = Enabled. The breakpoint module will only give a match if the ASID also matches.

www.BDTIC.com/ATMEL

9.4.6.7 Watchpoint Trigger

Table 9-32. 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

www.BDTIC.com/ATMEL

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.

www.BDTIC.com/ATMEL

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 9-33. Direct branch instructions

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

www.BDTIC.com/ATMEL

Table 9-34. 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 9-35. 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 9-36. 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.

www.BDTIC.com/ATMEL

.

Table 9-37. 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 9-38. 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

www.BDTIC.com/ATMEL

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 9-39. 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 9-40. 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

www.BDTIC.com/ATMEL

Resource Full messages can occur before the arrival of the message that the information belongs with.

Table 9-41. 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 9-42 for details.
6	TCODE	Fixed	Value = 27

Table 9-42. Resource Code (RCODE) description

Resource 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 9-43 will cause this message to be transmitted.

Table 9-43. 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 9-44.
6	TCODE	Fixed	Value = 33

Table 9-44. Event Code (EVCODE) description

Event Code (EVCODE)	**Event Description**
0b0000	Entry into Debug Mode
0b0001	Entry into Low Power Mode

www.BDTIC.com/ATMEL

Table 9-44. Event Code (EVCODE) description

Event Code (EVCODE)	Event Description
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 3 compliant Data Trace Unit. Both reads and writes can be traced. Information is traced between the CPU and data cache, which gives immediate access to modified data for cached memory accesses. This provides a direct correspondence between the CPU program and traced data, even if there may be a delay before the written cache data is actually flushed to the data memory.

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 125.

www.BDTIC.com/ATMEL

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 9-45. 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.

www.BDTIC.com/ATMEL

Table 9-46. 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 9-47. 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.

www.BDTIC.com/ATMEL

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 9-48. 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

9.6.4 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	27:20	Reserved	0	
R/W	19:12	ASID1	0	ASID to match for channel 1
R/W	11	ASID1EN	0	ASID1EN - ASID 1 enable
R/W	10:3	ASID0	0	ASID to match for channel 0

www.BDTIC.com/ATMEL

Table 9-49. Data Trace Control Register

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	2	ASID0EN	0	ASID1EN - ASID 0 enable
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.

www.BDTIC.com/ATMEL

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 Interface

9.8.1 Overview

The Memory Interface provides the debug tool with a mechanism for a DMA-like access to memory mapped resources both run-time and in Debug Mode. Memory mapped resources include main memory and memory mapped peripheral modules. Access to registers in peripheral modules may trigger unintended operation of the module, and the debug tool should generally include a list of access restrictions for the peripherals. The CPU register file is not memory mapped, and not accessible through the Memory Interface.

Note that the tool uses physical addresses when accessing memory mapped data through the Memory Interface, as opposed to Breakpoint/Watchpoints, which use virtual addresses.

www.BDTIC.com/ATMEL

9.8.2 Memory (Block) Access

The Memory Block Access is controlled by the registers Read/Write Access Data (RWD), Read/Write Access Address (RWA), and Read/Write Control/Status (RWCS). The tool accesses these registers via the JTAG port. A Memory Block Access provides the debug tool with a mechanism for a DMA-like access to memory mapped resources. In this document, the terms Memory Block Access and Memory Access refer to a memory block of any length from one single location (CNT = 1), to the full range supported by the OCD system (CNT = 16 383). When using the maximum size of word, the maximum memory that can be transferred in one block is thus 64KB.

9.8.2.1 Memory Read Operations

1. The tool writes RWA with physical address to be read.
2. The tool configures RWCS Register, including CNT, AC=1, and RW=0 to indicate (block) read.
3. The tool must wait until the data is ready from the MIU. If very slow memory is accessed the tool can check that the DV bit in RWCS is one before reading RWD.
4. The tool reads RWD. The MIU outputs the data and auto-increments the address.
5. Step 3 and 4 are repeated until the number of data specified by the CNT field has been read.
6. When the entire block has been read AC will be cleared by the MIU. RWA will point to the last location that was read. If the tool wishes to continue reading from this point RWCS must be written again.

9.8.2.2 Special cases:

- If the memory read operation results in an error, the ERR bit in RWCS will be set, and the data in RWD will not be valid (DV=0). AC will also be cleared.
- Any write to RWCS will abort an ongoing block access.

9.8.2.3 Memory Write Operations

1. The tool writes RWA with the first physical address to be written.
2. The tool configure the RWCS Register, including CNT, AC=1, and RW=1 to indicate (block) write.
3. The tool writes one data value to RWD.
4. The tool must wait until the MIU is ready to accept more data. This can be done by reading the ready bit in RWCS or by using the NEXUS-STATUS JTAG command.
5. Step 3 and 4 are repeated until the number of data specified by the CNT field has been transmitted.
6. When the entire block has been read AC will be cleared by the MIU. RWA will point to the last location that was written. If the tool wishes to continue reading from this point RWCS must be written again.

9.8.2.4 Special cases:

- If the memory write operation results in an error, the ERR bit in RWCS will be set and DV cleared. AC will also be cleared.
- Any write to RWCS will abort an ongoing block access.

www.BDTIC.com/ATMEL

9.8.3 Address Space

The SZ field of RWCS determines the word size (data type) of the memory access, while the CNT-field of the RWCS register determines the number of accesses of size SZ. Since Avr-32 is byte addressed, the accessed address range is from RWA through RWA + k * (CNT+1), where k is given by Table 9-54. The address in RWA must correspond to the Most Significant Byte of the data of size SZ in physical memory.

Note that the AVR32 uses a big-endian memory model. E.g., if the word located at address 0x0000 contains 0x12345678, the byte at 0x0000 will read 0x12, and the halfword at 0x0000 will read 0x1234.

Table 9-54. Address Increment as a Function of Word Size / Data Type

Access Type	SZ	$k = 2^{SZ}$
Byte access	000	1
Half-word access	001	2
Word access	010	4

9.8.4 Error Conditions

Errors during a memory access are indicated to the tool by setting the ERR and clearing the DV bit of the RWCS register.

There are 3 sources of errors:

1. The system bus signals an error
2. If the tool writes to the RWCS register during a single or block access, the access is terminated and indicated as an error.
3. The Tool writes SZ or CNT to an illegal value.

9.8.5 Data Cache operation

Memory accesses from the OCD system are served by the Data Cache. Since the Data Cache can buffer CPU accesses to memory, there may be an inconsistent data view between the cache and system memory. The OCD system will specify whether or not the access should be cached or uncached. By default, reads will be cached, and writes uncached. This can be altered by writing the Cache Control (CCTRL) bits in RWCS.

Uncached reads return the value in system memory, which may differ from the value in the cache if the CPU has written to this location. Cached reads return the value in the cache, i.e. the same as the value seen by the CPU. If the data is not present inside the cache, the Data Cache accesses the data through the system bus. Unlike a memory access from the CPU, a cache miss for an OCD access does not update the Data Cache memory or registers.

Uncached writes will change the value both in the cache and system memory. Cached writes will only change the cache value, and tag the cache line as dirty, ensuring it will be written to memory on the next cache flush. Cached writes are faster than uncached writes, but can cause temporary inconsistency between system and cache memory.

Access error indicated by the system bus results in an exception in the CPU if cached operation is used. When the error is due to an uncached memory access from the OCD System, the exception is not generated, but the Error bit is set in the Read Write Control Register (RWCS).

The Data Cache is a shared resource between the CPU and the OCD System. This resource is allocated solely to the CPU in normal operation. Hence, the OCD System will degrade perfor-

www.BDTIC.com/ATMEL

mance. This delay is particularly large for uncached accesses or cached accesses to locations not present in the cache.

9.8.6 NanoTrace

NanoTrace is an AVR32 specific debug feature which allows a JTAG-based emulator to observe limited trace information without the need for an AUX interface. Instead, the trace messages will be written to a circular buffer in a reserved space in the data memory, configured by the RWCS and RWA registers. NanoTrace employs the block write mechanism to write trace data to the internal SRAM, so block read/write is not available when NanoTrace is enabled. All messages normally written to the AUX port will be written to memory, so all kinds of trace, as well as watchpoint messages are written to the internal memory and can be reconstructed.

The AUX port does not need to be active for NanoTrace to function.

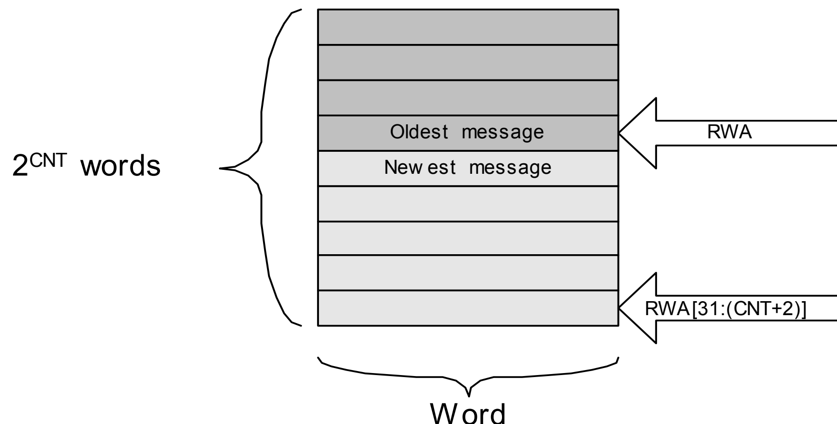
9.8.6.1 NanoTrace operation

To enable NanoTrace, the RWCS must be written with AC=1, RW=1, SZ=010, NTE=1 and CNT=2^n, where n is an integer between 0 and 28. The start address of the circular buffer must be written to RWA. The CNT field must be written to log2("size of buffer in bytes">>2), this restricts the size of the NanoTrace buffer to 2^n boundaries but permits up to 1 GB of trace buffer.

Once NanoTrace is enabled, messages are extracted frame by frame from the Transmit Queue and written to the RWD register. Only valid (i.e. non-idle) frames are extracted. When RWD has no room for more frames, it is written to the circular buffer in memory, as shown in Figure 9-11. The buffer is repeatedly overwritten with trace messages until NanoTrace is halted. This occurs when the NTE bit in RWCS is written to zero. Every time the buffer wraps, the next trace message is inserted with sync, to increase the portion of the trace buffer which can be uniquely reconstructed.

When NanoTrace is halted, the block read/write mechanism can again be used to access memory locations from the debugger.

Figure 9-11. NanoTrace memory arrangement.



9.8.6.2 Extracting NanoTrace messages

When NanoTrace is halted, or no more trace messages are generated (e.g. in OCD Mode), the RWA register will point to the word following the last message written to memory. If the circular buffer has been completely filled and thus overwritten at least once, the RWCS:WRAPPED bit

www.BDTIC.com/ATMEL

will be set. This means that the word pointed to by RWA is part of the oldest message. If RWCS:WRAPPED is cleared, only the messages from RWA[31:(CNT+2)] to RWA-4 contain valid message data.

The trace log can thus be reconstructed by reading words from RWA (or RWS[31:(CNT+2)] if RWCS:WRAPPED is cleared) to RWA-4 in the circular RAM buffer. When reaching the address RWA+CNT*4, the address should be wrapped down to RWA[31:(CNT+2)]. Frames consist of the value of the MSEO pins in the most significant bit positions, and the value of the MDO pins in the least significant bit positions. Frames are aligned to the most significant bit within each word, as shown in Figure 9-12.

Since RWD is only written to the buffer when a whole word of data is filled, the last frames of the last message may not have been transmitted to memory. RWCS:DV will be set to indicate that RWD contains valid trace data, and these frames can be extracted by reading RWD. Empty frame positions within RWD are tagged as "Idle", i.e. MSEO = 0b11.

Figure 9-12 shows an example of a NanoTrace buffer, with RWA starting at 0x1000 and CNT = 10 (i.e. the buffer size is 1024 words, or 4096 frames). When the trace was stopped, RWCS:WRAPPED is set and RWA = 0x1234, so the last word of frame data written to the memory is located at 0x1230, and a partially filled word is in RWD. In this example, the last message (shown in white) in the Transmit Queue was an Indirect Branch message with Sync. The same example was shown for regular AUX port transmission. The last two frames of the message still reside in RWD, which has been only partially filled.

Figure 9-12. Frame organization within a word.

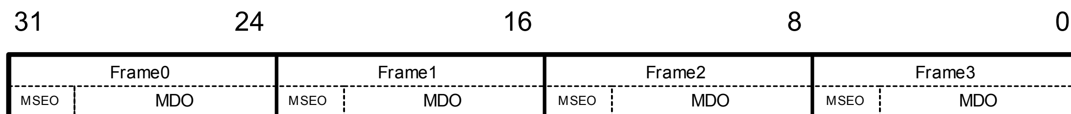
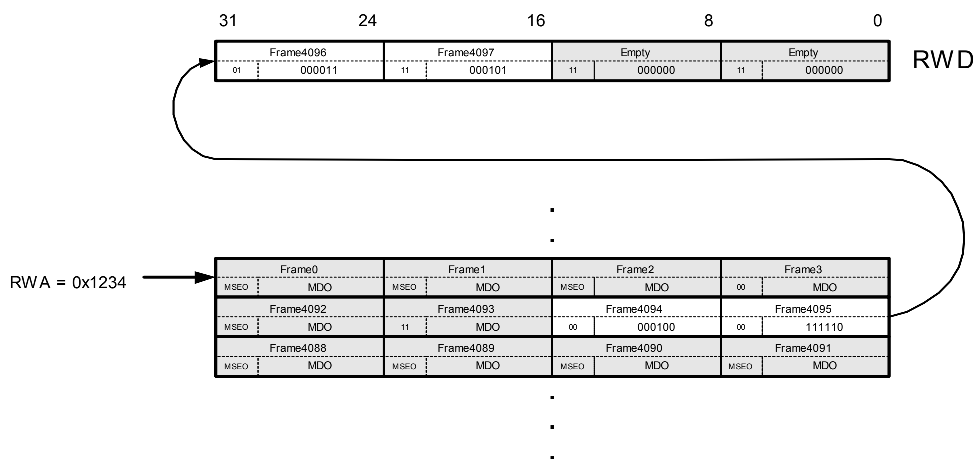


Figure 9-13. Reconstructing a NanoTrace message.



9.8.6.3 NanoTrace access protection

If the CPU attempts to write the data memory reserved for NanoTrace messages, the CPU software or message reconstruction can fail. To automatically detect this source of error, it is possible to write the NanoTrace Access Protection (NTAP) bit in RWCS to one. This will cause a

www.BDTIC.com/ATMEL

hardware error to be triggered if the CPU attempts to access the protected area. This allows the emulator to abort the program execution and notify the user about the illegal access.

NanoTrace access protection will only function correctly when physical and virtual addresses are the same for the memory region reserved for NanoTrace. If this is not the case, NTAP should stay zero to avoid incorrect access error breakpoints.

Note that NanoTrace access protection will never trigger in Monitor Mode.

9.8.6.4 *Overrun control*

The DC:OVC bits works for NanoTrace as well as for AUX port messages. However, the overrun prevention will not be as efficient for NanoTrace. If the Transmit Queue becomes full, the CPU will not issue any more instructions, but already issued instructions will be allowed to complete. If these instructions generate trace information, the Transmit Queue may overrun even when the CPU is stalled.

9.8.6.5 *NanoTrace Buffer Control*

By default, the NanoTrace buffer will be repeatedly overwritten until NanoTrace is stopped. However, by writing the RWCS:NTBC bits, it is possible to control the behavior when the buffer becomes full. In this case, RWD will not contain trace information, and does not need to be read out. RWA will point to the first address in the buffer, so RWA does not need to be rewritten if NanoTrace is restarted.

In some cases, only the first trace messages after NanoTrace is enabled are interesting. In this case, NanoTrace can be disabled when the buffer is full. The debugger will detect that this has occurred by observing when RWCS:NTE is negated. RWCS:AC and DV will also be cleared, to indicate that the memory operation is complete, and no valid trace information exists in RWD. To restart NanoTrace, RWCS:NTE and AC must be written to one.

Alternatively, Debug Mode can be triggered when the buffer is full. This will set the NanoTrace Buffer Full bit in the Development Status register (DS:NTBF). RWCS:NTE will stay set, but AC and DV will be cleared. The debugger can then read out the NanoTrace buffer in Debug Mode, before restarting execution. To restart NanoTrace when exiting Debug Mode, RWCS:NTE and RWCS:AC must be written to one.

9.8.6.6 *CRC-32 check of a memory block*

The memory interface unit can generate a CRC-32 checksum on a memory block.

The standard CRC-32 (802.3) polynomial is used:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

To enable this feature the debugger must set the AC=1, CRC=1, SIZE=2 (word) and CNT=<number of words in block> bit in RWCS and the start of the block in RWA. The MIU will then read the memory block and put a CRC-32 of the memory block in RWD when AC is cleared and DV is set. The debugger can continue the CRC generation on a new block by rewriting the RWCS with AC, CRC, SIZE and CNT when a CRC block is finished and the CRC bit is still set. The CRC in RWD after the second block will be CRC32(block1 + block2).

9.8.7 Messages

The Memory Interface generates no messages, all features are accessed with regular read / write messages on the JTAG interface.

www.BDTIC.com/ATMEL

9.8.8 Registers

9.8.8.1 Read/Write Access Control/Status (RWCS) Register

Table 9-55. *Read/Write Access Control/Status (RWCS)*

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	31	AC	0	AC - Access 0 = No access ongoing 1 = Start access
R/W	30	RW	0	RW - Memory Access Read/Write 0 = Read 1 = Write
R/W	29:27	SZ	0	SZ - Data Size 000 = Byte 001 = Half-Word 010 = Word 011 = Reserved 1xx = Reserved
R/W	26:25	CCTRL	00	CCTRL - Cache Control 00 = Auto 01 = Always use cached memory view 10 = Always use uncached memory view 11 = Reserved
R/W	24	WRAPPED	0	WRAPPED - NanoTrace Buffer wrapped Indicates that the RWA pointer to the nanotrace buffer has wrapped at least once.
R/W	23	NTAP	0	NTAP - NanoTrace Access Protection Enables NanoTrace access protection.
R/W	22	NTE	0	NTE - NanoTrace Enable Enables NanoTrace.
R/W	21:20	NTBC	00	NTBC - NanoTrace Buffer Control 00 = Overwrite buffer 01 = Disable trace when buffer full 10 = Trigger breakpoint when buffer full 11 = Reserved
R/W	19	CRC	0	CRC - CRC Enable Enables CRC of memory area.
R	18:16	-	0	Reserved
R/W	15:2	CNT	0	CNT - Access Count Number of accesses of word size SZ.CNT is an unsigned number.
R	1	ERR	0	Last access generated an error
R	0	DV	0	Data Valid in RWD

www.BDTIC.com/ATMEL

AC

The tool writes the AC bit to one to initiate an access. The AC field is negated by the MIU upon completion of the access requested by the tool. Any write operation to the RWCS register will terminate any access in process, including the remaining of the block access. If the write operation sets AC=1, the previous (block) access will be terminated, but a new one will be initiated.

SZ

SZ determines the access size. The bits are written to by the tool.

RW

RW determines whether the access is a read or write. The RW bit is written by the tool.

NTE

Enable nanotrace. When NanoTrace is enabled, trace messages will be written to the data memory.

CRC

When this bit is set the MIU will read the entire memory area specified with RWA and CNT and place a CRC-32 signature of this area in RWD when AC is cleared and DV is set. NTE and CRC is mutually exclusive, SZ must be word. When the CRC generation of a block is complete the CRC-32 will be in RWD. If the tool wishes to continue calculating CRC beyond the first block it must rewrite RWCS with AC=1, CRC=1, SZ=10 and appropriate CNT.

WRAPPED

This bit is set when the RWA pointer into the NanoTrace buffer has wrapped at least once. The emulator should reset this bit when a new NanoTrace session is started.

CCTRL

MIU memory access is routed through the Data Cache. There are two ways of accessing the data cache, cached and uncached. The safest way of accessing the memory is using cached reads and uncached writes, the Auto setting of CCTRL automatically uses this configuration.

Note that when the Auto setting is used with NanoTrace, the MIU will write to cached memory to improve trace performance.

In the cached memory view writes will be write back, and any errors will be routed to the CPU as bus error, the ERR bit will not be set. Reads will access the cache and see the CPU's view of the memory.

In the uncached memory view writes will be write through, but they will update the cache to preserve memory consistency any bus errors will be reported back to the OCD and ERR bit will be set. Reads will go straight to the bus and bypass any cache buffers. In this mode the memory view may be different from the CPU's view of the memory.

CNT

To request a block move, CNT is set by the tool to the number of accesses of data size SZ, zero is an illegal value. The CNT field is incremented by the OCD system during an in-progress block move. When CNT wraps to 0, the block move is complete, and the OCD system negates the AC field. If an error occurs, CNT indicates how far the block access had progressed before the error occurred.

DV and ERR

www.BDTIC.com/ATMEL

If errors occur, the target will terminate the access, including any remaining block accesses, within one access cycle of the target. In this case, the access in progress when the RWCS Register is written is not guaranteed to complete. Errors are either due to errors on the system bus during an access requested by the tool, triggered by writing the RWCS Register while any single or block access is in progress, or attempting a block access with CNT=0. See Table 9-56 for a description.

Note that for Read Accesses, DV is always cleared when RWD is read, including for the last access.

Table 9-56. Read/Write Access Status Bit Encoding

DV	ERR	Read Action	Write Action
0	0	Read Access has not completed	Write Access completed without error
0	1	Read Access error has occurred	Write Access error has occurred
1	0	Read Access completed without error	Write Access has not completed
1	1	Not Allowed	Not Allowed

9.8.8.2 *Read/Write Access Address (RWA) Register*

The RWA Register is used by the tool to program the physical address of memory mapped resource to be accessed, or the lowest physical address (i.e. lowest unsigned value) for a block access (CNT>0). RWA must correspond to the most significant byte of the data of size SZ. Refer to "Address Space" on page 143 for a description of the address range during a Memory Block Access..

Table 9-57. *Read/Write Access Address (RWA)*

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	31:0	RWA	0x0000_0 000	Physical address to be accessed

9.8.8.3 *Read/Write Access Data (RWD) Register*

The RWD Register contains the data to be written for the next Memory Block Write access, and the read data for completed memory read accesses.

Note that the data is presented in little-endian format in the RWD register as shown in Table 9-59.

Table 9-58. Organization of RWD for Different Data Sizes

Access	31 24	23 16	15 8	7 0
Byte				Byte
Half-word			MS Byte	LS Byte
Word	MS Byte			LS Byte

www.BDTIC.com/ATMEL

.

Table 9-59. Read/Write Data (RWD)

R/W	Bit Number	Field Name	Init. Val.	Description
R/W	31:0	RWD	0x0000 _0000	32 bits of data read from a physical address location or to be written to a physical address location.

9.9 OCD Message Summary

Table 9-60. Message Summary

TCODE	Message	Public / Vendor Defined	Page
0	Debug Status (DEBS)	Public	page 102
1	Reserved		
2	Ownership Trace (OT)	Public	page 141
3	Program Trace, Direct Branch (PTDB)	Public	page 131
4	Program Trace, Indirect Branch (PTIB)	Public	page 132
5	Data Trace, Data Write (DTDW)	Public	page 137
6	Data Trace, Data Read (DTDR)	Public	page 138
7	Reserved		
8	Error (ERROR)	Public	page 117
9	Program Trace Synchronization (PTSY)	Public	page 133
10	Reserved		
11	Program Trace, Direct Branch with Sync (PTDBS)	Public	page 134
12	Program Trace, Indirect Branch with Sync (PTIBS)	Public	page 134
13	Data Trace, Data Write with Sync (DTDWS)	Public	page 137
14	Data Trace, Data Read with Sync (DTDRS)	Public	page 138
15	Watchpoint Hit (WH)	Public	page 125
16–26	Reserved		
27	Program Trace Resource Full (PTRF)	Public	page 134
28–32	Reserved		
33	Program Trace Correlation (PTC)	Public	page 135
34–55	Reserved		
56	Trace Watchpoint Hit (TWH)	Vendor	page 125
57	Direct Branch with Target Address (DBTA)	Vendor	page 131
58-62	Reserved	Vendor	
63 (0x3F)	Vendor Defined Extension Message Reserved	Vendor	

www.BDTIC.com/ATMEL

Table 9-60 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 111.

Table 9-61 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

	Message format			
Nexus Message	**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, 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]**
Watchpoint Hit	15	WPHIT[7:0]	-	-
Trace Watchpoint Hit	56	WPHIT[1:0]	-	-

www.BDTIC.com/ATMEL

9.10 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 111).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 9-62. OCD Register Summary

Register Index	mtdr/mfdr index	Register	Access Type	Page
0	0	Device ID (DID)	R	page 103
1	4	Reserved	—	
2	8	Development Control (DC)	R/W	page 105
3	12	Reserved	—	
4	16	Development Status (DS)	R	page 107
5-6	20-24	Reserved	—	
7	28	Read/Write Access Control/Status (RWCS)	R/W	page 147
8	32	Reserved	—	
9	36	Read/Write Access Address (RWA)	R/W	page 149
10	40	Read/Write Access Data (RWD)	R/W	page 149
11	44	Watchpoint Trigger (WT)	R/W	page 129
12	48	Reserved	—	
13	52	Data Trace Control (DTC)	R/W	page 139
14–15	56-60	Data Trace Start Address (DTSA) Channel 0 to 1	R/W	page 140
16-17	64-68	Reserved	—	
18–19	72-76	Data Trace End Address (DTEA) Channel 0 to 1	R/W	page 140
20-21	80-84	Reserved	—	
22	88	PC Breakpoint/Watchpoint Control 0A (BWC0A)	R/W	page 126
23	92	PC Breakpoint/Watchpoint Control 0B (BWC0B)	R/W	page 126
24	96	PC Breakpoint/Watchpoint Control 1A (BWC1A)	R/W	page 126
25	100	PC Breakpoint/Watchpoint Control 1B (BWC1B)	R/W	page 126
26	104	PC Breakpoint/Watchpoint Control 2A (BWC2A)	R/W	page 126
27	108	PC Breakpoint/Watchpoint Control 2B (BWC2B)	R/W	page 126
28	112	Data Breakpoint/Watchpoint Control 3A (BWC3A)	R/W	page 128
29	116	Data Breakpoint/Watchpoint Control 3B (BWC3B)	R/W	page 128
30	120	PC Breakpoint/Watchpoint Address 0A (BWA0A)	R/W	page 125
31	124	PC Breakpoint/Watchpoint Address 0B (BWA0B)	R/W	page 125
32	128	PC Breakpoint/Watchpoint Address 1A (BWA1A)	R/W	page 125
33	132	PC Breakpoint/Watchpoint Address 1B (BWA1B)	R/W	page 125

www.BDTIC.com/ATMEL

Table 9-62. OCD Register Summary

Register Index	mtdr/mfdr index	Register	Access Type	Page
34	136	PC Breakpoint/Watchpoint Address 2A (BWA2A)	R/W	page 125
35	140	PC Breakpoint/Watchpoint Address 2B (BWA2B)	R/W	page 125
36	144	Data Breakpoint/Watchpoint Address 3A (BWA3A)	R/W	page 127
37	148	Data Breakpoint/Watchpoint Address 3B (BWA3B)	R/W	page 127
38	152	Breakpoint/Watchpoint Data 3A (BWD3A)	R/W	page 127
39	156	Breakpoint/Watchpoint Data 3B (BWD3B)	R/W	page 127
40–65	160-260	Reserved	—	
64	256	Nexus Configuration (NXCFG)	R	page 103
65	260	Debug Instruction Register (DINST)	R/W	page 109
66	264	Debug Program Counter (DPC)	R/W	page 109
67	268	CPU Control Mask	R/W	
68	272	Debug Communication CPU Register (DCCPU)	R/W	page 104
69	276	Debug Communication Emulator Register (DCEMU)	R/W	page 104
70	280	Debug Communication Status Register (DCSR)	R/W	page 105
71	284	Ownership Trace Process ID (PID)	R/W	page 141
72-74	288-296	Reserved	—	
75	300	Event Pair Control 3 (EPC3)	R/W	page 127
76	304	AUX port Control (AXC)	R/W	page 118
77– 255	308-1020	Reserved	—	

www.BDTIC.com/ATMEL

10. Instruction cycle summary

This chapter presents the grouping of the instructions in the AVR32 architecture. All the instructions in each group behave similarly in the pipeline, and are discussed as a group in the rest of this documentation.

10.1 Validity of timing information

This chapter presents information about the timing requirements of each instruction. This information should be used together with measurements from cycle-correct simulations. Issues like branch prediction, data hazards, cache misses and exceptions may cause the cycle requirements of real implementations to differ from the theoretical number presented here.

All timing presented here represents best case numbers. The following factors are assumed:

- No data hazards are experienced
- No resource conflicts are encountered in the pipeline
- All data and instruction accesses hit in the caches, and no protection violations are experienced

10.2 Definitions

The following definitions are used in the tables below:

10.2.1 Issue

An instruction is *issued* when it leaves the IS stage and enters the M1, A1, or DA stage.

10.2.2 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 to the same subpipe. Generally, an instruction has an issue latency of one if the following instruction is issued to another subpipe and no data hazards exist.

10.2.3 Result latency

The *result latency* represents the number of cycles between the issue of the instruction and the availability of the result from the forwarding logic. Some instructions, like 64-bit multiplications, produce several results. For these instructions, the result latency for both the first part of the result and the last part of the result are presented. After the result latency period, the data is available for forwarding, and instructions with data dependencies may execute.

10.2.4 Flag latency

The *flag latency* represents the number of clock cycles required between the issue of an instruction updating the flags and the issue of another instruction using the flags. Note that flags are also forwarded, in most cases making the flags available to the following instruction. As an example, for an *add* followed by a branch, the branch will read the flags updated by the *add*. No stall is required between the *add* and the branch.

www.BDTIC.com/ATMEL

10.3 Special considerations

10.3.1 PC as destination register

Most instructions can use PC as destination register. This will result in a jump to the calculated address. Forwarding is not implemented, so jumping is performed when the target address is available in WB.

10.3.2 Branch prediction

Branch prediction allows the branch penalty to be removed for correctly predicted branches. For erroneously predicted branches, a branch delay of four cycles is imposed. For correctly predicted, folded branches, the branch executes in zero cycles. Erroneously predicted folded branches execute in four cycles.

Table 10-1. Predicted branch and call cycle requirement

Instruction	Predicted correctly	Predicted erroneously	Folded correctly	Folded erroneously	Not predicted
br disp	1	4	0	4	4
rjmp disp	1	4	0	4	4
rcall disp	1	4	NA	NA	4

10.3.3 Return address stack

A return address stack is implemented, allowing the subprogram return address to be available early. The return address stack can keep 4 elements. If more elements are pushed, the oldest element is overwritten. Hardware keeps control over the number of valid elements on the stack. Stack over- and underflow is handled automatically by hardware, at the cost of performance loss. When a return is attempted with an empty return address stack, the return instruction is considered as not predicted.

Table 10-2. Return instruction cycle requirement

Instruction	Predicted correctly	Predicted erroneously	Not predicted
ret, cond != AL	1	4	4
ret, cond == AL	2	-	4
mov PC, LR	2	-	4
popm with PC in reglist	2	-	6
ldm with PC in reglist	2	-	6

www.BDTIC.com/ATMEL

10.4 ALU Operations

This group comprises simple single-cycle ALU operations like add and sub. The conditional sub and mov instructions are also in this group. All instructions in this group take one cycle to execute, and the result is available for use by the following instruction.

Table 10-3. Timing of ALU operations

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

www.BDTIC.com/ATMEL

Table 10-3. Timing of ALU operations

subhh.w	C	Rd, Rx<part>, Ry<part>	Subtract signed halfwords (32 ← 16 - 16)	1	1	1
sub{cond4}	E	Rd, imm	Subtract immediate if condition satisfied.	1	1	1
tnbz	C	Rd	Test no byte equal to zero.	1	1	1
and	C	Rd, Rs	Logical AND.	1	1	1
	E	Rd, Rx, Ry << sa		1	1	1
	E	Rd, Rx, Ry >> sa		1	1	1
andn	C	Rd, Rs	Logical AND NOT.	1	1	1
andh	E	Rd, imm	Logical AND High Halfword, low halfword is unchanged.	1	1	1
	E	Rd, imm, COH	Logical AND High Halfword, clear other halfword.	1	1	1
andl	E	Rd, imm	Logical AND Low Halfword, high halfword is unchanged.	1	1	1
	E	Rd, imm, COH	Logical AND Low Halfword, clear other halfword.	1	1	1
com	C	Rd	One's Complement (NOT).	1	1	1
eor	C	Rd, Rs	Logical Exclusive OR.	1	1	1
	E	Rd, Rx, Ry << sa		1	1	1
	E	Rd, Rx, Ry >> sa		1	1	1
eorh	E	Rd, imm	Logical Exclusive OR (High Halfword).	1	1	1
eorl	E	Rd, imm	Logical Exclusive OR (Low Halfword).	1	1	1
or	C	Rd, Rs	Logical (Inclusive) OR.	1	1	1
	E	Rd, Rx, Ry << sa		1	1	1
	E	Rd, Rx, Ry >> sa		1	1	1
orh	E	Rd, imm	Logical OR (High Halfword).	1	1	1
orl	E	Rd, imm	Logical OR (Low Halfword).	1	1	1
tst	C	Rd, Rs	Test register for zero.	1	1	1
bfins	E	Rd, Rs, o5, w5	Insert the lower w5 bits of Rs in Rd at bit-offset o5.	1	1	1

www.BDTIC.com/ATMEL

Table 10-3. Timing of ALU operations

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

www.BDTIC.com/ATMEL

Table 10-3. Timing of ALU operations

mov{cond4}	E	Rd, Rs	Copy register if condition is true.	1	1	1
	E	Rd, imm	Load immediate into register if condition is true.	1	1	1
csrf	C	b5	Clear status register flag.	1	1	1
csrfcz	C	b5	Copy status register flag to C and Z.	1	1	1
ssrf	C	b5	Set status register flag.	1	1	1
sr{cond4}	C	Rd	Conditionally set register to true or false.	1	1	1

10.5 Multiply16 operations

These instructions require one pass through the multiplier array and produce a 32-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. The Q flag is a sticky flag, so subsequent instructions will not stall due to Q flag dependencies.

Table 10-4. Timing of Multiply16 operations

Mnemonics		Operands	Description	Issue latency	Result latency	Flag latency
mul	E	Rd, Rs, imm	Multiply immediate.	1	2	N/A
mulhh.w	E	Rd, Rx<part>, Ry<part>	Signed Multiply of halfwords. (32 ← 16 x 16)	1	2	N/A
mulnhh.w	E	Rd, Rx<part>, Ry<part>	Signed Multiply of halfwords. (32 ← 16 x 16)	1	2	N/A
mulnwh.d	E	Rd, Rx, Ry<part>	Signed Multiply, word and halfword. (48 ← 32 x 16)	1	2 + delayed wb	N/A
mulwh.d	E	Rd, Rx, Ry<part>	Signed Multiply, word and halfword. (48 ← 32 x 16)	1	2 + delayed wb	N/A
mulsathh.h	E	Rd, Rx<part>, Ry<part>	Fractional signed multiply with saturation. Return halfword. (16 ← 16 x 16)	1	2	Q: 3
mulsathh.w	E	Rd, Rx<part>, Ry<part>	Fractional signed multiply with saturation. Return word. (32 ← 16 x 16)	1	2	Q: 3

www.BDTIC.com/ATMEL

Table 10-4. Timing of Multiply16 operations

mulsatwh.w	E	Rd, Rx, Ry<part>	Fractional signed multiply with saturation. Return word. (32 ← 32 x 16)	1	2	Q: 3
mulsatrndhh.h	E	Rd, Rx<part>, Ry<part>	Signed multiply with rounding. Return halfword. (16 ← 16 x 16)	1	2	Q: 3
mulsatrndwh.w	E	Rd, Rx, Ry<part>	Signed multiply with rounding. Return halfword. (32 ← 32 x 16)	1	2	Q: 3

10.6 Mac16 operations

These instructions require one pass through the multiplier array and produce a 32-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.

The *machh.d* and *macwh.d* instruction uses a 48-bit accumulator. The accumulator in the MUL pipeline is wide enough to perform an 48-bit accumulation in a single cycle. The requirements for *machh.d* and *macwh.d* is listed separately below. In these two instructions, the high part of the result is written back first, contrary to the other doubleword instructions. The low part of the result is written back when the MUL write port is idle. This implies that other MUL instructions may complete before the low part of a *machh.d* or *macwh.d* is written back. Hardware interlocks are present in order to guarantee correct execution in this case, guaranteeing that no hazards will occur.

This group does not set any flags, except for the *macsat* instruction which set Q if saturation occurred. The Q flag is a sticky flag, so subsequent instructions will not stall due to Q flag dependencies. If saturation occurred, the Q flag is set after 3 or 4 cycles, depending on an accumulator cache hit.

www.BDTIC.com/ATMEL

Table 10-5. Timing of Mac16 operations

Mnemonics		Operands	Description	Issue latency	Result latency	Flag latency
machh.w	E	Rd, Rx<part>, Ry<part>	Multiply signed halfwords and accumulate. (32 ← 16x16 + 32)	1/2	2/3	N/A
machh.d	E	Rd, Rx<part>, Ry<part>	Multiply signed halfwords and accumulate. (48 ← 16x16 + 48)	1/2	2/3 + delaye d wb	N/A
macwh.d	E	Rd, Rx, Ry<part>	Multiply signed word and halfword and accumulate. (48 ← 32 x 16 + 48)	1/2	2/3 + delaye d wb	N/A
macsathh.w	E	Rd, Rx<part>, Ry<part>	Fractional signed multiply accumulate with saturation. Return word. (32 ← 16 x 16 + 32)	1/2	2/3	Q: 3/4

10.7 MulMac32 operations

These instructions require two passes through the multiplier array to produce a 32-bit result. For *mac*, 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 10-6. Timing of MulMac32 operations

Mnemonics		Operands	Description	Issue latency	Result latency	Flag latency
mac	E	Rd, Rx, Ry	Multiply accumulate. (32 ← 32x32 + 32)	2/3	3/4	N/A
mul	E	Rd, Rx, Ry	Multiply. (32 ← 32 x 32)	2	3	N/A

10.8 MulMac64 operations

These instructions require two passes through the multiplier array to produce a 64-bit result. For *macs* and *macu*, 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. The low

www.BDTIC.com/ATMEL

part of the result is written back 1 cycle before the high part, and the result latencies presented are for the low part of the result.

Table 10-7. Timing of MulMac64 operations

Mnemonics		Operands	Description	Issue latency	Result latency	Flag latency
macs.d	E	Rd, Rx, Ry	Multiply signed accumulate. (64 ← 32x32 + 64)	3/4	4/5	N/A
macu.d	E	Rd, Rx, Ry	Multiply unsigned accumulate. (64 ← 32x32 + 64)	3/4	4/5	N/A
muls.d	E	Rd, Rx, Ry	Signed Multiply. (64 ← 32 x 32)	3	4	N/A
mulu.d	E	Rd, Rx, Ry	Unsigned Multiply. (64 ← 32 x 32)	3	4	N/A

10.9 Divide operations

These instructions require several cycles in the multiply pipeline to complete. The quotient (Q) is written back 1 cycle before the remainder (R).

Table 10-8. Timing of divide operations

Mnemonics		Operands	Description	Issue latency	Result latency	Flag latency
divs	E	Rd, Rx, Ry	Divide signed. (32 ← 32/32) (32 ← 32%32)	33	Q:33 R:34	N/A
divu	E	Rd, Rx, Ry	Divide unsigned. (32 ← 32/32) (32 ← 32%32)	33	Q:33 R:34	N/A

10.10 Saturate operations

The saturate instructions use both the A1 and A2 stages to produce a valid result. Flags are forwarded so that they are ready for the following instruction to use.

Table 10-9. Timing of saturate operations

Mnemonics		Operands	Description	Issue latency	Result latency	Flag latency
satadd.h	E	Rd, Rx, Ry	Saturated add halfwords.	1	2	1
satadd.w	E	Rd, Rx, Ry	Saturated add.	1	2	1
satsub.h	E	Rd, Rx, Ry	Saturated subtract halfwords.	1	2	1
satsub.w	E	Rd, Rx, Ry	Saturated subtract.	1	2	1
	E	Rd, Rs, imm		1	2	1

www.BDTIC.com/ATMEL

Table 10-9. Timing of saturate operations (Continued)

satrnds	E	Rd >> sa, b5	Signed saturate from bit given by sa after a right shift with rounding of b5 bit positions.	1	2	1
satrndu	E	Rd >> sa, b5	Unsigned saturate from bit given by sa after a right shift with rounding of b5 bit positions.	1	2	1
sats	E	Rd >> sa, b5	Shift sa positions and do signed saturate from bit given by b5.	1	2	1
satu	E	Rd >> sa, b5	Shift sa positions and do unsigned saturate from bit given by b5.	1	2	1

10.11 Load and store operations

This group includes all the load and store instructions. The LS pipeline has a dedicated adder with an operand shift functionality, which performs all the address calculations except the ones needed for indexed addressing. The additions needed in indexed addressing is performed by the adder in the A1 stage. The A1 adder also performs the writeback address calculation for autoincrement and autodecrement operation.

Loaded word data are available directly after the D pipestage. Byte and halfword data must be extended and rotated before they are valid. This is performed in the WB stage. *Ldins* and *ldswp* instructions also require modification in the WB stage before their results are valid. *Stswp* instructions require modification before their data is output to the cache. This modification is performed in the D stage. All store instructions may experience write-after-read hazards, and therefore subsequent instructions writing to the register to be stored are stalled until the store instruction has left the D stage.

Load of unaligned word addresses will increase the issue and result latency with one or two cycles, depending on the alignment. Store of unaligned word addresses will increase the issue latency with one or two cycles, depending on the alignment. Load of word-aligned doubleword will increase the issue and result latency with one cycle. Store of word-aligned doubleword will increase the issue latency with one cycle.

Table 10-10. Timing of load and store operations

Mnemonics		Operands	Description.	Issue latency	Result latency	Flag latency
ld.ub	C	Rd, Rp++	Load unsigned byte with post-increment.	1	3	N/A
	C	Rd, --Rp	Load unsigned byte with pre-decrement.	1	3	N/A
	C	Rd, Rp[disp]	Load unsigned byte with displacement.	1	3	N/A
	E	Rd, Rp[disp]		1	3	N/A
	E	Rd, Rb[Ri<<sa]	Indexed Load unsigned byte.	1	3	N/A

www.BDTIC.com/ATMEL

Table 10-10. Timing of load and store operations (Continued)

ld.sb	E	Rd, Rp[disp]	Load signed byte with displacement.	1	3	N/A
	E	Rd, Rb[Ri<<sa]	Indexed Load signed byte.	1	3	N/A
ld.uh	C	Rd, Rp++	Load unsigned halfword with post-increment.	1	3	N/A
	C	Rd, --Rp	Load unsigned halfword with pre-decrement.	1	3	N/A
	C	Rd, Rp[disp]	Load unsigned halfword with displacement.	1	3	N/A
	E	Rd, Rp[disp]		1	3	N/A
	E	Rd, Rb[Ri<<sa]	Indexed Load unsigned halfword.	1	3	N/A
ld.sh	C	Rd, Rp++	Load signed halfword with post-increment.	1	3	N/A
	C	Rd, --Rp	Load signed halfword with pre-decrement.	1	3	N/A
	C	Rd, Rp[disp]	Load signed halfword with displacement.	1	3	N/A
	E	Rd, Rp[disp]		1	3	N/A
	E	Rd, Rb[Ri<<sa]	Indexed Load signed halfword.	1	3	N/A
ld.w	C	Rd, Rp++	Load word with post-increment.	1	2	N/A
	C	Rd, --Rp	Load word with pre-decrement.	1	2	N/A
	C	Rd, Rp[disp]	Load word with displacement.	1	2	N/A
	E	Rd, Rp[disp]		1	2	N/A
	E	Rd, Rb[Ri<<sa]	Indexed Load word.	1	2	N/A
	E	Rd, Rp[Ri<part> << 2]	Load word with extracted index.	1	2	N/A
ld.d	C	Rd, Rp++	Load doubleword with post-increment.	1	2	N/A
	C	Rd, --Rp	Load doubleword with pre-decrement.	1	2	N/A
	C	Rd, Rp	Load doubleword.	1	2	N/A
	E	Rd, Rp[disp]	Load double with displacement.	1	2	N/A
	E	Rd, Rb[Ri<<sa]	Indexed Load double.	1	2	N/A
ldins.b	E	Rd<part>, Rp[disp]	Load byte with displacement and insert at specified byte location in Rd.	1	3	N/A

www.BDTIC.com/ATMEL

Table 10-10. Timing of load and store operations (Continued)

ldins.h	E	Rd<part>, Rp[disp]	Load halfword with displacement and insert at specified halfword location in Rd.	1	3	N/A
ldswp.sh	E		Load halfword with displacement, swap bytes and sign-extend	1	3	N/A
ldswp.uh	E	Rd, Rp[disp]	Load halfword with displacement, swap bytes and zero-extend	1	3	N/A
ldswp.w	E		Load word with displacement and swap bytes.	1	3	N/A
lddpc	C	Rd, PC[disp]	Load with displacement from PC.	1	2	N/A
lddsp	C	Rd, SP[disp]	Load with displacement from SP.	1	2	N/A
st.b	C	Rp++, Rs	Store with post-increment.	1	1	N/A
	C	--Rp, Rs	Store with pre-decrement.	1	1	N/A
	C	Rp[disp], Rs	Store byte with displacement.	1	1	N/A
	E	Rp[disp], Rs		1	1	N/A
	E	Rb[Ri<<sa], Rs	Indexed Store byte.	1	1	N/A
st.d	C	Rp++, Rs	Store with post-increment.	1	1	N/A
	C	--Rp, Rs	Store with pre-decrement.	1	1	N/A
	C	Rp, Rs	Store doubleword	1	1	N/A
	E	Rp[disp], Rs	Store double with displacement	1	1	N/A
	E	Rb[Ri<<sa], Rs	Indexed Store double.	1	1	N/A
st.h	C	Rp++, Rs	Store with post-increment.	1	1	N/A
	C	--Rp, Rs	Store with pre-decrement.	1	1	N/A
	C	Rp[disp], Rs	Store halfword with displacement.	1	1	N/A
	E	Rp[disp], Rs		1	1	N/A
	E	Rb[Ri<<sa], Rs	Indexed Store halfword.	1	1	N/A

ATMEL®
www.BDTIC.com/ATMEL

Table 10-10. Timing of load and store operations (Continued)

st.w	C	Rp++, Rs	Store with post-increment.	1	1	N/A
	C	--Rp, Rs	Store with pre-decrement.	1	1	N/A
	C	Rp[disp], Rs	Store word with displacement.	1	1	N/A
	E	Rp[disp], Rs		1	1	N/A
	E	Rb[Ri<<sa], Rs	Indexed Store word.	1	1	N/A
stcond	E	Rp[disp], Rs	Conditional store with displacement.	1	1	N/A
stdsp	C	SP[disp], Rs	Store with displacement from SP.	1	1	N/A
sthh.w	E	Rp[disp<<2], Rx, Ry	Combine halfwords to word and store with displacement	1	1	N/A
	E	Rb[Ri<<sa], Rx, Ry	Combine halfwords to word and store indexed	1	1	N/A
stswp.h	E	Rp[disp], Rs	Swap bytes and store halfword with displacement.	1	1	N/A
stswp.w	E		Swap bytes and store word with displacement.	1	1	N/A

10.12 Load and store multiple operations

These instructions perform multiple data accesses. The writeback pointer is calculated by the A1 adder if needed, and the incremental pointer addresses are generated by the DA adder under control by the LS pipeline control FSM. This FSM enables the LS pipe to operate decoupled from the rest of the pipeline.

As the table shows, the updated pointer is available after the instruction has left the A1 stage. If PC is specified for a load, and the return stack is empty, a 6 cycle penalty is taken, as the pipeline must be flushed. If enough registers are specified in the register list, this PC load penalty will be masked by the regular register loads.

Store multiple instructions have the same write-after-write hazard detection as regular store instructions. Subsequent instructions writing to a register that is to be stored are stalled until the store has left the D stage.

www.BDTIC.com/ATMEL

LDM and POPM have a flag latency of 2 cycles.

Table 10-11. Timing of load and store multiple operations

Mnemonics		Operands	Description	Pointer update ready	First loaded data ready	Penalty for PC load
ldm	E	Rp{++}, Reglist16	Load multiple registers. R12 is tested if PC is loaded.	1	2	+6
ldmts	E	Rp{++}, Reglist16	Load multiple registers in application context for task switch.	1	2	+6
popjc	C		Pop Java context from frame	1	2	-
popm	C	Reglist8	Pop multiple registers from stack. R12 is tested if PC is popped.	1	2	+6
pushjc	C		Push Java context to frame	1	-	-
pushm	C	Reglist8	Push multiple registers to stack.	1	-	-
stm	E	{--}Rp, Reglist16	Store multiple registers.	1	-	-
stmts	E	{--}Rp, Reglist16	Store multiple registers in application context for task switch.	1	-	-

10.13 Branch operations

The branch instructions are subject to branch prediction. This implies that the latencies related to branches depends on whether the prefetch unit correctly predicted the outcome of the branch, and if it had time to prefetch the branch target. The *rjmp* instruction is unconditional, and always taken. It can never be predicted incorrectly.

The *ret* instruction has dedicated return stack hardware. The return address of call instructions is pushed on a 4-entry loop stack. When a *ret* instruction is encountered and predicted taken, the top stack element is popped and the instruction fetches are redirected to this address. In a way, *ret* behaves very similarly to branches, except that their target address is fetched from a loop stack when predicted taken.

www.BDTIC.com/ATMEL

Table 10-12. Timing of branch operations

Mnemonics		Operands	Description	Predicted correctly	Predicted incorrectly	Predictable
br{cond3}	C	disp	Branch if condition satisfied.	See chapter 10.3.2 and chapter 10.3.3		
br{cond4}	E	disp				
rjmp	C	disp	Relative jump.			
ret{cond4}	C	Rs	Conditional return from subroutine with move and test of return value.			

10.14 Call operations

Call instructions behave similarly to branches, except that the link register must be updated. Branches can therefore never be reduced to zero cycles. The relative branches and *acall* are always predicted, and can never be predicted incorrectly. The other call instructions are never predicted, and will therefore have to flow through the pipeline. *Mcall* and *acall* will flow through the pipeline, and the loaded target address is not ready until the WB pipestage. All correctly predicted instructions take 1 or 2 cycles, depending on their size and alignment.

Table 10-13. Timing of call operations

Mnemonics		Operands / Syntax	Description	Predicted correctly	Predicted incorrectly	Predictable
acall	C	disp	Application call.	-	6	No
icall	C	Rd	Register indirect call.	-	4	No
mcall	E	Rp[disp]	Memory call.	-	6	No
rcall	C	disp	Relative call.	1/2	4	Yes
	E	disp		1/2	4	Yes
scall	C		Supervisor call.	-	4	No
breakpoint	C		Breakpoint.	-	4	No

10.15 Return from exception operations

These instructions are never predicted, but flow through the pipe as regular instructions. The target address is calculated when the instruction is in the A1 stage. In the following cycle, the target instruction is fetched, and the execution stream continues from there.

Table 10-14. Timing of return from exception operations

Mnemonics		Operands / Syntax	Description	Issue latency	Result latency	Flag latency
retd	C		Return from debug mode	4	N/A	N/A
rete	C		Return from exception	4	N/A	N/A
rets	C		Return from supervisor call	4	N/A	N/A

www.BDTIC.com/ATMEL

10.16 Swap operation

The swap instruction perform two atomical memory accesses, first one read and then one write. Write-after-write hazards may arise for the store part of *xchg*. This will stall subsequent instructions writing to the register to store until the store part of *xchg* has left D.

Table 10-15. Timing of swap operation

Mnemonics		Operands / Syntax	Description.	Issue latency	Result latency	Flag latency
xchg	E	Rd, Rx, Ry	Exchange register and memory	2	3	N/A

10.17 System register operations

This group moves data to and from the system registers. Forwarding and hazard detection is implemented for the system registers. Latencies vary depending on where the system register being is placed in the system, refer to Table 2-2 on page 10 for details. Accesses to system registers in A1 take one cycle. Accesses to registers on the TCB bus have a read latency of four cycles, and writes have an issue latency of one cycle. Special care must be taken to avoid hazards when using the some of these instructions, refer to Section 3.9 on page 27 for details.

Table 10-16. Timing of system register operations

Mnemonics		Operands / Syntax	Description.	Issue latency	Result latency	Flag latency
mfdr	E	Rd, SysRegNo	Move debug register to Rd.	2	4	N/A
mfsr	E	Rd, SysRegNo	Move system register to Rd.	1/2	1/4	N/A
mtdr	E	SysRegNo, Rs	Move Rs to debug register.	1	4	N/A
mtsr	E	SysRegNo, Rs	Move Rs to system register.	1	4	N/A
musfr	C	Rs	Move Rs to status register.	1	2	N/A
mustr	C	Rd	Move status register to Rd.	1	2	N/A
tlbr	C		Read TLB entry.	1	3	N/A
tlbs	C		Search TLB for entry.	1	3	N/A
tlbw	C		Write TLB entry.	1	3	N/A

www.BDTIC.com/ATMEL

10.18 System control operations

This group contains simple single-cycle instructions that control the behaviour of different parts of the system. Special care must be taken to avoid hazards when using the *cache* instruction, refer to Section 3.9.5 on page 29 for details.

Table 10-17. Timing of system control operations

Mnemonics		Operands / Syntax	Description.	Issue latency	Result latency	Flag latency
cache	E	Rp[disp], Op5	Perform cache operation	1	1	N/A
frs	C		Invalidate the return address stack	1	1	N/A
pref	E	Rp[disp]	Prefetch cache line	1	1	N/A
sleep	E	Op8	Enter SLEEP mode.	1	1	N/A
sync	E	Op8	Flush write buffer	1	1	N/A

10.19 Coprocessor operations

Figure 10-1. Timing of coprocessor operations

Mnemonics		Operands / Syntax	Description.	Issue latency	Result latency	Flag latency
cop	E	CP#, CRd, CRx, CRy, Op	Coprocessor operation.	1	-	N/A
ldc.d	E	CP#, CRd, Rp[disp]	Load coprocessor register.	1	5	N/A
	E	CP#, CRd, --Rp	Load coprocessor register with pre-decrement.			
	E	CP#, CRd, Rb[Ri<<sa]	Load coprocessor register with indexed addressing.			
ldc0.d	E	CRd, Rp[disp]	Load coprocessor 0 register.	1	5	N/A
ldc.w	E	CP#, CRd, Rp[disp]	Load coprocessor register.	1	5	N/A
	E	CP#, CRd, --Rp	Load coprocessor register with pre-decrement.			
	E	CP#, CRd, Rb[Ri<<sa]	Load coprocessor register with indexed addressing.			
ldc0.w	E	CRd, Rp[disp]	Load coprocessor 0 register.	1	5	N/A
ldcm.d	E	CP#, Rp{++}, ReglistCPD8	Load multiple coprocessor registers.	As LDM	As LDM	N/A
ldcm.w	E	CP#, Rp{++}, ReglistCPH8	Load multiple coprocessor registers.	As LDM	As LDM	N/A

www.BDTIC.com/ATMEL

Figure 10-1. Timing of coprocessor operations (Continued)

ldcm.w	E	CP#, Rp{++}, ReglistCPL8	Load multiple coprocessor registers.	As LDM	As LDM	N/A
mvcr.d	E	CP#, Rd, CRs	Move from coprocessor to register.	2	4	N/A
mvcr.w	E	CP#, Rd, CRs	Move from coprocessor to register.	2	4	N/A
mvrc.d	E	CP#, CRd, Rs	Move from register to coprocessor.	1	5	N/A
mvrc.w	E	CP#, CRd, Rs	Move from register to coprocessor.	1	5	N/A
stc.d	E	CP#, Rp[disp], CRs	Store coprocessor register.	1	5	N/A
	E	CP#, Rp++, CRs	Store coprocessor register with post-increment.			
	E	CP#, Rb[Ri<<sa], CRs	Store coprocessor register with indexed addressing.			
stc0.d	E	Rp[disp], CRs	Store coprocessor 0 register.	1	5	N/A
stc.w	E	CP#, Rp[disp], CRs	Store coprocessor register.	1	5	N/A
	E	CP#, Rp++, CRs	Store coprocessor register with post-increment.			
	E	CP#, Rb[Ri<<sa], CRs	Store coprocessor register with indexed addressing.			
stc0.w	E	Rp[disp], CRs	Store coprocessor 0 register.	1	5	N/A
stcm.d	E	CP#, {--}Rp, ReglistCPD8	Store multiple coprocessor registers.	As STM +1	As STM +1	N/A
stcm.w	E	CP#, {--}Rp, ReglistCPH8	Store multiple coprocessor registers.	As STM +1	As STM +1	N/A
stcm.w	E	CP#, {--}Rp, ReglistCPL8	Store multiple coprocessor registers.	As STM +1	As STM +1	N/A

10.20 Java return operation

Table 10-18. Timing of Java return operation

Mnemonics		Operands / Syntax	Description.	Issue latency	Result latency	Flag latency
retj	C		Return from Java trap.	4	N/A	N/A

www.BDTIC.com/ATMEL

10.21 SIMD operations

This group comprises instructions operating on multiple data in parallel. Some instructions in this group take one cycle to execute, and the result is available for use by the following instruction. Other instructions perform saturation in A2, and need two cycles before the result is ready.

Table 10-19. Timing of SIMD Operations

Mnemonics		Operands / Syntax	Description.	Issue latency	Result latency	Flag latency
pabs.{sb/sh}	E	Rd, Rs	Packed Absolute Value.	1	1	1
packsh.{ub/sb}	E	Rd, Rx, Ry	Pack Halfwords to Bytes.	1	1	1
packw.sh	E	Rd, Rx, Ry	Pack Words to Halfwords.	1	1	1
padd.{b/h}	E	Rd, Rx, Ry	Packed Addition.	1	1	1
paddh.{ub/sh}	E	Rd, Rx, Ry	Packed Addition with halving.	1	1	1
padds.{ub/sb/uh/sh}	E	Rd, Rx, Ry	Packed Addition with Saturation.	1	2	1
paddsub.h	E	Rd, Rx<part>, Ry<part>	Packed Halfword Addition and Subtraction.	1	1	1
paddsubh.sh	E	Rd, Rx<part>, Ry<part>	Packed Halfword Addition and Subtraction with halving.	1	1	1
paddsubs.{uh/sh}	E	Rd, Rx<part>, Ry<part>	Packed Halfword Addition and Subtraction with Saturation.	1	2	1
paddx.h	E	Rd, Rx, Ry	Packed Halfword Addition with Crossed Operand.	1	1	1
paddxh.sh	E	Rd, Rx, Ry	Packed Halfword Addition with Crossed Operand and Halving.	1	1	1
paddxs.{uh/sh}	E	Rd, Rx, Ry	Packed Halfword Addition with Crossed Operand and Saturation.	1	2	1
pasr.{b/h}	E	Rd, Rs, {sa}	Packed Arithmetic Shift Left.	1	1	1
pavg.{ub/sh}	E	Rd, Rx, Ry	Packed Average.	1	1	1
plsl.{b/h}	E	Rd, Rs, {sa}	Packed Logic Shift Left.	1	1	1
plsr.{b/h}	E	Rd, Rs, {sa}	Packed Logic Shift Right.	1	1	1
pmax.{ub/sh}	E	Rd, Rx, Ry	Packed Maximum Value.	1	1	1
pmin.{ub/sh}	E	Rd, Rx, Ry	Packed Minimum Value.	1	1	1
psad	E	Rd, Rx, Ry	Sum of Absolute Differences.	1	2	2
psub.{b/h}	E	Rd, Rx, Ry	Packed Subtraction.	1	1	1
psubadd.h	E	Rd, Rx<part>, Ry<part>	Packed Halfword Subtraction and Addition.	1	1	1

www.BDTIC.com/ATMEL

Table 10-19. Timing of SIMD Operations

psubaddh.sh	E	Rd, Rx<part>, Ry<part>	Packed Halfword Subtraction and Addition with halving.	1	1	1
psubadds.{uh/sh}	E	Rd, Rx<part>, Ry<part>	Packed Halfword Subtraction and Addition with Saturation.	1	2	1
psubh.{ub/sh}	E	Rd, Rx, Ry	Packed Subtraction with halving.	1	1	1
psubs.{ub/sb/uh/sh}	E	Rd, Rx, Ry	Packed Subtraction with Saturation.	1	2	1
psubx.h	E	Rd, Rx, Ry	Packed Halfword Subtraction with Crossed Operand.	1	1	1
psubxh.sh	E	Rd, Rx, Ry	Packed Halfword Subtraction with Crossed Operand and Halving.	1	1	1
psubxs.{uh/sh}	E	Rd, Rx, Ry	Packed Halfword Subtraction with Crossed Operand and Saturation.	1	2	1
punpck{ub/sb}.h	E	Rd, Rs<part>	Unpack Bytes to Halfwords.	1	1	1

www.BDTIC.com/ATMEL

11. Glossary

The following abbreviations and terms are used in this document.

Term	Definition
Recoverable Exception	An exception that saves enough state so that normal program execution can resume after the exception routine has finished.
Processor Consistency	A strict processor consistency is maintained if only recoverable exceptions can occur. Otherwise, the processor has a weak consistency.
Instruction Commit	An instruction is said to be committed when it has updated the processor state.
Processor State	The processor state is comprised of the following modules: • The register file • The status register • The system registers • The coprocessors • The MMU • The debug hardware
Contaminated instruction	An instruction flowing through the pipeline that has somehow caused an exception. If such an instruction is about to commit, the exception routine will be entered. An example of a contaminated instruction is an instruction that caused a protection violation when it was fetched. Contaminated instructions will not always generate exceptions, they may for example be flushed from the pipe by branches further down the pipe.
BHT	Branch History Table
BTB	Branch Target Buffer
HUM	See Hit under miss
Icache	Instruction cache
Dcache	Data cache
Frozen instruction	Instruction halted in a pipeline stage du to some kind of data hazard
Nexus	The IEEE-ISTO 5001™-2003 debug standard for embedded processors.
OCD	On-Chip Debug
AUX	The Nexus-defined Auxiliary port for trace debug information.
API	Application Program Interface
JTAG	Joint Test Action Group, i.e. IEEE1149.1 standard
FCU	Flow Control Unit
MIU	Memory Interface Unit
JVM	Java Virtual Machine
Debug Mode	A CPU mode dedicated to executing instructions for debug purposes.
Monitor Mode	Debug Mode running from instructions fetched from memory.
OCD Mode	Debug Mode running from instructions entered by an external debugger.

www.BDTIC.com/ATMEL

12. Revision History

12.1 Rev. 32001A-06/06

1. **Initial version.**

www.BDTIC.com/ATMEL

www.BDTIC.com/ATMEL

Table of contents

1 Introduction 2

1.1 The AVR family 2

1.2 The AVR32 Microprocessor Architecture 2

1.3 Event handling 3

1.4 Java Support 3

1.5 Microarchitectures 4

1.6 The AVR32 AP implementation 5

2 Programming Model 6

2.1 Architectural compatibility 6

2.2 Implementation options 6

2.3 Register file configuration 6

2.4 Status register configuration 7

2.5 System registers 10

2.6 Configuration Registers 16

3 Pipeline 21

3.1 Overview 21

3.2 Prefetch unit 21

3.3 Decode unit 22

3.4 ALU pipeline 22

3.5 Multiply pipeline 23

3.6 Load-store pipeline 24

3.7 Writeback 25

3.8 Forwarding hardware and hazard detection 25

3.9 Hazards not handled by the hardware 27

3.10 Event handling 30

3.11 Entry points for events 32

3.12 Interrupt latencies 46

3.13 Processor consistency 47

4 Virtual memory 48

4.1 Memory map 48

4.2 Understanding the MMU 50

4.3 Operation of the MMU and MMU exceptions 60

5 Prefetch Unit 63

www.BDTIC.com/ATMEL

5.1 Instruction buffer 63
5.2 Branch prediction 64

6 Instruction Cache 69

6.1 Behaviour 69
6.2 Cache operations 70
6.3 Memory coherency 71
6.4 Debug access to ICache memories 72

7 Data Cache and Write Buffer 74

7.1 Data cache behaviour 74
7.2 Write buffer behaviour 75
7.3 Cache and write buffer operations 75
7.4 Prefetch instruction 77
7.5 Sync instructions 77
7.6 Memory mapped cache memories 77

8 Coprocessor interface 79

8.1 Coprocessor pipeline 79
8.2 TCB specification 79
8.3 Connecting coprocessors to the TCB bus 82
8.4 Execution of coprocessor instructions 82
8.5 Timing diagrams 84

9 OCD system 86

9.1 Overview 86
9.2 CPU Development Support 90
9.3 Debug Port 111
9.4 Breakpoints 119
9.5 Program trace 130
9.6 Data Trace 136
9.7 Ownership Trace 140
9.8 Memory Interface 141
9.9 OCD Message Summary 150
9.10 OCD Register Summary 152

10 Instruction cycle summary 154

10.1 Validity of timing information 154
10.2 Definitions 154

www.BDTIC.com/ATMEL

10.3 Special considerations 155
10.4 ALU Operations 156
10.5 Multiply16 operations 159
10.6 Mac16 operations 160
10.7 MulMac32 operations 161
10.8 MulMac64 operations 161
10.9 Divide operations 162
10.10 Saturate operations 162
10.11 Load and store operations 163
10.12 Load and store multiple operations 166
10.13 Branch operations 167
10.14 Call operations 168
10.15 Return from exception operations 168
10.16 Swap operation 169
10.17 System register operations 169
10.18 System control operations 170
10.19 Coprocessor operations 170
10.20 Java return operation 171
10.21 SIMD operations 172

11 Glossary 174

12 Revision History 175

12.1 Rev. 32001A-06/06 175

www.BDTIC.com/ATMEL