ARM11™ MPCore™ Processor

Revision: r0p3

Technical Reference Manual

ARM DDI 0360C

ARM11 MPCore Processor
Technical Reference Manual

Release Information

Change history

Date	Issue	Confidentiality	Change
02 February 2005	A	NonConfidential	First release
14 September 2005	B	NonConfidential	Name change, minor corrections and update. Section 1.3.1 conditional execution statement amended, BTAC flushing in relation to FCSE PID and Context ID clarified, and in Section 3.4.24 and Section 3.4.25. Section 1.6 Configurable options now mentions wrapper options in tables 1-3 and 1-4. Section 2.11.4 STREXB in Figure 2-11, values corrected. Section 2.11.5 STREXH in Figure 2-12, values corrected. Section 3.4.4 CPU ID section corrections. Table 3-41 Performance Monitoring Events, new line added: Event number 0x16 Event definition STB deadlock condition detected Section 7.2.6 Note added about STB deadlock condition. Section 8.1.1 Cross-reference to AXI Specification clarified. Section 11.5.5 Signal names spelled out. Table 19-8 values corrected. RVI appendix added. AXI appendix added. Drain Write Buffer replaced by Data Synchronization Barrier throughout.
16 December 2005	C	NonConfidential	Minor corrections. See *Product revisions* on page 1-32 for details.

Confidentiality Status

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents

ARM11 MPCore Processor Technical Reference Manual

List of Tables

ARM11 MPCore Processor Technical Reference Manual

List of Figures
ARM11 MPCore Processor Technical Reference Manual

Preface

This preface introduces the *ARM11 MPCore™ Processor r0p3 Technical Reference Manual*. It contains the following sections:

- *About this book* on page xxiv
- *Feedback* on page xxx.

About this book

This document is the *Technical Reference Manual* (TRM) for the ARM11 MPCore Processor.

Product revision status

The r*n*p*n*v*n* identifier indicates the revision status of the product described in this document, where:

r*n*	Identifies the major revision of the product.
p*n*	Identifies the minor revision or modification status of the product.

Intended audience

This manual is written for hardware and software engineers implementing ARM11 MPCore processor system designs. It provides information to enable designers to integrate the processor into a target system as quickly as possible.

Note

The ARM11 MPCore Processor is a single IP core. It consists of between one and four Central Processing Units (CPUs). Individual CPUs are referred to as MP11 CPUs.

Using this manual

This manual is divided into:

Part A *ARM11 MPCore Processor*

This part contains information on the ARM11 MPCore processor.

Part B *Vector Floating-Point*

This part contains information on the *Vector Floating-point Processor* (VFP).

Part C *Appendices*

This part contains information common to the ARM11 MPCore processor and the VFP.

List of chapters

The following list contains a more detailed description of the contents of each part:

Part A ARM11 MPCore processor

Chapter 1 ***Introduction***
Read this chapter for an introduction to the ARM11 MPCore processor and descriptions of the major functional blocks.

Chapter 2 ***Programmer's Model***
Read this chapter for a description of the MPCore registers and programming details.

Chapter 3 ***Control Coprocessor CP15***
Read this chapter for a description of the MPCore control coprocessor CP15 registers and programming details.

Chapter 4 ***Unaligned and Mixed-Endian Data Access Support***
Read this chapter for a description of the unaligned and mixed-endian data access support.

Chapter 5 ***Memory Management Unit***
Read this chapter for a description of the MPCore *Memory Management Unit* (MMU) and the address translation process.

Chapter 6 ***Program Flow Prediction***
Read this chapter for a description of the functions of the MPCore Prefetch Unit, including static and dynamic branch prediction and the return stack.

Chapter 7 ***Level 1 Memory System***
Read this chapter for a description of the MPCore level one memory system, including caches, TLBs, and Write Buffer.

Chapter 8 ***Level 2 Memory System***
Read this chapter for a description of the MPCore level two memory system, including supported AXI transfers, CPU synchronization, and synchronization operations.

Chapter 9 ***MPCore Private Memory Region***
Read this chapter for a description of the Coherency Protocol and the programmable registers of the *Snoop Control Unit* (SCU).

Chapter 10 ***MPCore Distributed Interrupt Controller***
Read this chapter for a description of the Distributed Interrupt Controller.

Chapter 11 ***Clocking, Resets, and Power Management***
Read this chapter for a description of the MPCore clocking modes and the reset signals.

Chapter 12 ***Debug***
Read this chapter for a description of the MPCore debug support.

Chapter 13 ***Debug Test Access Port***
Read this chapter for a description of the JTAG-based MPCore Debug Test Access Port.

Chapter 14 ***Cycle Timings and Interlock Behavior***
Read this chapter for a description of the MPCore instruction cycle timing and for details of the interlocks.

Part B ***Vector Floating-Point***
Read this for a description of the VFP.

Chapter 15 ***Introduction to VFP***
Read this chapter for an introduction to the VFP.

Chapter 16 ***VFP Register File***
Read this chapter for a description of the VFP register file.

Chapter 17 ***VFP Programmer's Model***
Read this chapter for a description of the VFP programmer's model.

Chapter 18 ***VFP Instruction Execution***
Read this chapter for a description of forwarding, hazards, and parallel execution in the VFP instruction pipelines.

Chapter 19 ***VFP Exception Handling***
Read this chapter for a description of VFP exception handling.

Part C ***Appendices***
Read this appendix for a description of the MPCore signals.

Appendix A ***Signal Descriptions***
Read this appendix for a description of the MPCore signals.

Appendix B ***AC Characteristics***
Read this appendix for a description of the MPCore AC characteristics.

Appendix C ***MBIST Controller and Dispatch Unit***
Read this appendix for a description of the MBIST Controller and Dispatch Unit.

Appendix D ***Scan chain ordering with RVI***

Read this appendix for a description of the scan chain ordering with RVI.

Appendix E ***IEM***
Read this appendix for a description of the Intelligent Energy Manager wrappers.

Glossary Read the Glossary for definitions of terms used in this manual.

Conventions

Conventions that this manual can use are described in:

- *Typographical*
- *Timing diagrams*
- *Signals* on page xxviii
- *Numbering* on page xxix.

Typographical

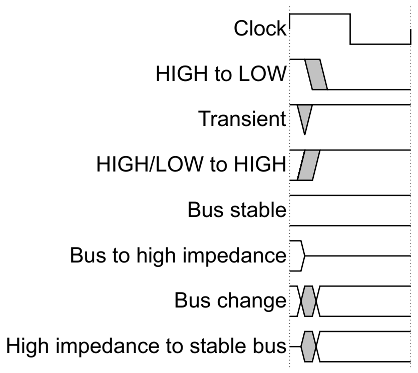
The typographical conventions are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace` (mono underlined)	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
< and >	Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. They appear in normal font in running text. For example: • `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>` • The Opcode_2 value selects which register is accessed.

Timing diagrams

The figure named *Key to timing diagram conventions* on page xxviii explains the components used in these diagrams. When variations occur they have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.



Key to timing diagram conventions

Signals

The signal conventions are:

Signal level	The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means HIGH for active-HIGH signals and LOW for active-LOW signals.
Prefix A	Denotes *Advanced eXtensible Interface* (AXI) global and address channel signals.
Prefix B	Denotes AXI write response channel signals.
Prefix C	Denotes AXI low-power interface signals.
Prefix H	Denotes *Advanced High-performance Bus* (AHB) signals.
Prefix n	Denotes active-LOW signals except in the case of AXI, AHB, or *Advanced Peripheral Bus* (APB) reset signals.
Prefix P	Denotes APB signals.
Prefix R	Denotes AXI read channel signals.
Prefix W	Denotes AXI write channel signals.
Suffix n	Denotes AXI, AHB, and APB reset signals.

Numbering

The numbering convention is:

<size in bits>'<base><number>

This is a Verilog method of abbreviating constant numbers. For example:

- 'h7B4 is an unsized hexadecimal value.
- 'o7654 is an unsized octal value.
- 8'd9 is an eight-bit wide decimal value of 9.
- 8'h3F is an eight-bit wide hexadecimal value of `0x3F`. This is equivalent to b00111111.
- 8'b1111 is an eight-bit wide binary value of b00001111.

Further reading

This section lists publications by ARM Limited, and by third parties.

ARM Limited periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets, addenda, and the Frequently Asked Questions list.

ARM publications

This manual contains information that is specific to the ARM11 MPCore processor. See the following documents for other relevant information:

- *ARM11 MPCore Processor Implementation Guide* (ARM DII 0126)
- *ARM11 MPCore Processor Configuration Manual* (ARM DII 0127)
- *L220 Cache Controller Technical Reference Manual* (ARM DDI 0329)
- *AMBA AXI Protocol v1.0 Specification* (ARM IHI 0022)
- *ARM Architecture Reference Manual Edition 3* (ARM DDI 0100)
- *Jazelle V1 Architecture Reference Manual* (ARM DDI 0225)
- *RealView Compilation Tools Developer Guide* (ARM DUI 0203)
- *RealView ICE User Guide* (ARM DUI 0155)
- *Intelligent Energy Controller Technical Overview* (ARM DTO 0005).

Feedback

ARM Limited welcomes feedback on both the ARM11 MPCore processor and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on this manual

If you have any comments on this manual, send email to `errata@arm.com` giving:

- the title
- the number
- the relevant page number(s) to which your comments apply
- a concise explanation of your comments.

ARM Limited also welcomes general suggestions for additions and improvements.

Part A

ARM11 MPCore™ Processor

Chapter 1
Introduction

This chapter introduces the ARM11 MPCore processor and its features. It contains the following sections:

1.1 About the ARM11 MPCore processor

The ARM11 MPCore processor incorporates up to four MP11 CPUs that implement the ARM architecture v6K. It supports the ARM and Thumb instruction sets, Jazelle technology to enable direct execution of Java bytecodes, and a range of SIMD DSP instructions that operate on 16-bit or 8-bit data values in 32-bit registers.

The ARM11 MPCore processor is a high-performance, low-power, ARM cached multiprocessor macrocell that provides full virtual memory capabilities.

The ARM11 MPCore processor features:

Up to four MP11 CPUs

- a *Snoop Control Unit* (SCU) responsible for maintaining coherency among MP11 CPUs L1 data caches
- a Distributed Interrupt Controller with support for legacy ARM interrupts
- a private timer and a private watchdog per MP11 CPU
- AXI high-speed *Advanced Microprocessor Bus Architecture* (AMBA) level two interfaces.

Each MP11 CPU features:

- an integer unit with integral EmbeddedICE-RT logic
- an eight-stage pipeline
- branch prediction with return stack
- coprocessors 14 and 15
- Instruction and Data *Memory Management Units* (MMUs), managed using MicroTLB structures backed by a unified Main TLB
- Instruction and data caches, including a non-blocking data cache with *Hit-Under-Miss* (HUM)
- the data cache is physically indexed, physically tagged
- the data cache is write back, write allocate only
- the instruction cache is virtually indexed, physically tagged
- 32-bit instruction interface and 64-bit interface to the data cache
- hardware support for data cache coherency
- *Vector Floating-Point* (VFP) coprocessor support
- JTAG-based debug.

Figure 1-1 on page 1-3 shows the main blocks of the ARM11 MPCore processor.

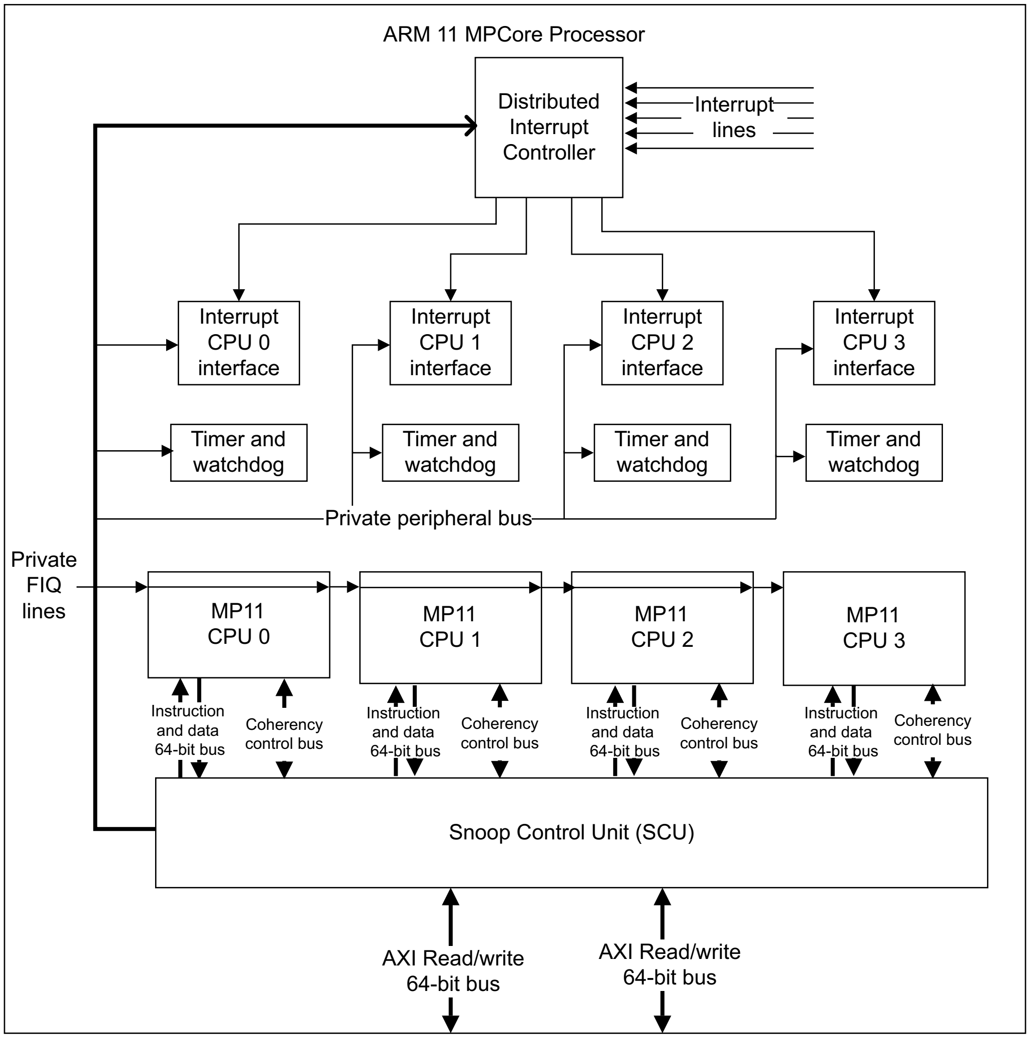

ARM 11 MPCore Processor
Distributed Interrupt Controller
Interrupt lines
Interrupt CPU 0 interface
Interrupt CPU 1 interface
Interrupt CPU 2 interface
Interrupt CPU 3 interface
Timer and watchdog
Timer and watchdog
Timer and watchdog
Timer and watchdog
Private peripheral bus
Private FIQ lines
MP11 CPU 0
MP11 CPU 1
MP11 CPU 2
MP11 CPU 3
Instruction and data 64-bit bus
Coherency control bus
Instruction and data 64-bit bus
Coherency control bus
Instruction and data 64-bit bus
Coherency control bus
Instruction and data 64-bit bus
Coherency control bus
Snoop Control Unit (SCU)
AXI Read/write 64-bit bus
AXI Read/write 64-bit bus

Figure 1-1 ARM11 MPCore processor block diagram

1.2 Extensions to ARMv6

The MPCore Multiprocessor provides support for extensions to ARMv6 that include:

- Store and Load Exclusive instructions for bytes, halfwords and doublewords, and a new Clear Exclusive instruction.
- A WaitForInterrupt instruction in the ARM and Thumb instruction set
- New WaitForEvent and SendEvent instructions
- A true no-operation instruction and yield instruction.
- Architectural remap registers.
- Revised use of TEX remap bits. The ARMv6 MMU page table descriptors use a large number of bits to describe all of the options for inner and outer cachability. In reality, it is believed that no application will need all of these options simultaneously. Therefore it is possible to configure the MP11 CPUs to support only a small number of options by means of the TEX remap mechanism. This implies a level of indirection in the page table mappings.

  The TEX CB encoding table provides two OS managed page table bits. For binary compatibility with existing ARMv6 ports of OSs, this gives a separate mode of operation of the MMU. This is called the TEX Remap configuration and is controlled by bit [28] TR in CP15 Register 1.
- Revised use of AP bits. In the MP11 CPUs the APX and AP[1:0] encoding b111 is Privileged or User mode read only access. AP[0] indicates an abort type, Access Bit fault, when CP15 c1[29] is 1.

1.3 MP11 CPU overview

The main blocks of each MP11 CPU are:

- *Integer core*
- *Load Store Unit (LSU)* on page 1-9
- *Prefetch unit* on page 1-9
- *Memory system* on page 1-9.

1.3.1 Integer core

The MP11 CPUs are built around an ARM11 integer core in an ARMv6 implementation that runs the 32-bit ARM, 16-bit Thumb, and 8-bit Jazelle instruction sets. The integer core contains EmbeddedICE-RT logic and a JTAG debug interface to enable hardware debuggers to communicate with the processor. The integer core is described in more detail in the following sections:

- *Instruction set categories*
- *Conditional execution* on page 1-6
- *Registers* on page 1-6
- *Modes and exceptions* on page 1-6
- *Thumb instruction set* on page 1-6
- *DSP instructions* on page 1-6
- *Media extensions* on page 1-7
- *Datapath* on page 1-7
- *Branch prediction* on page 1-8
- *Return stack* on page 1-9.

Instruction set categories

The instruction sets are divided into four categories:

- data processing instructions
- load and store instructions
- branch instructions
- coprocessor instructions.

Note

Only load, store, and swap instructions can access data from memory.

Conditional execution

The processor conditionally executes nearly all ARM instructions. You can decide if the condition code flags, Negative, Zero, Carry, and Overflow, are updated according to their result.

Registers

MP11 CPUs contain:

- 31 general-purpose 32-bit registers
- seven dedicated 32-bit registers.

Note

At any one time, 16 registers are visible. The remainder are banked registers used to speed up exception processing.

Modes and exceptions

The core provides a set of operating and exception modes, to support systems combining complex operating systems, user applications, and real-time demands. There are seven operating modes, five of which are exception processing modes:

- User mode
- Supervisor mode
- fast interrupt
- normal interrupt
- memory aborts
- software interrupts
- Undefined instruction.

Thumb instruction set

Thumb is an extension to the ARM architecture. It contains a subset of the most commonly-used 32-bit ARM instructions that has been encoded into 16-bit wide opcodes, to reduce memory requirements.

DSP instructions

The ARM DSP instruction set extensions provide the following:

- 16-bit data operations
- saturating arithmetic
- MAC operations.

Multiply instructions are processed using a single-cycle 32x16 implementation. There are 32x32, 32x16, and 16x16 multiply instructions (MAC).

Media extensions

The ARMv6 instruction set provides media instructions to complement the DSP instructions. The media instructions are divided into the following main groups:

- Additional multiplication instructions for handling 16-bit and 32-bit data, including dual-multiplication instructions that operate on both 16-bit halves of their source registers.

  This group includes an instruction that improves the performance and size of code for multi-word unsigned multiplications.

- Instructions to perform *Single Instruction Multiple Data* (SIMD) operations on pairs of 16-bit values held in a single register, or on quadruplets of 8-bit values held in a single register. The main operations supplied are addition and subtraction, selection, pack, and saturation.

- Instructions to extract bytes and halfwords from registers and zero-extend or sign-extend them. These include a parallel extraction of two bytes followed by extension of each byte to a halfword.

- Instructions to perform the unsigned *Sum-of-Absolute-Differences* (SAD) operation. This is used in MPEG motion estimation.

Datapath

The datapath consists of three pipelines:
- ALU/shift pipe
- MAC pipe
- load-store pipe, see *Load Store Unit (LSU)* on page 1-9.

ALU/shift pipe

The ALU/shift pipeline executes most of the ALU operations, and includes a 32-bit barrel shifter. It consists of three pipeline stages:

Shift The Shift stage contains the full barrel shifter. All shifts, including those required by the LSU, are performed in this stage.

The saturating left shift, which doubles the value of an operand and saturates it, is implemented in the Shift stage.

ALU The ALU stage performs all arithmetic and logic operations, and generates the condition codes for instructions that set these operations.

The ALU stage consists of a logic unit, an arithmetic unit, and a flag generator. Evaluation of the flags is performed in parallel with the main adder in the ALU. The flag generator is enabled only on flag-setting operations.

To support the DSP instructions, the carry chains of the main adder are divided to enable 8 and 16-bit SIMD instructions.

Sat The Sat stage implements the saturation logic required by the various classes of DSP instructions.

MAC pipe

The MAC pipeline executes all of the enhanced multiply, and multiply-accumulate instructions.

The MAC unit consists of a 32x16 multiplier plus an accumulate unit, which is configured to calculate the sum of two 16x16 multiplies. The accumulate unit has its own dedicated single register read port for the accumulate operand.

To minimize power consumption, each of the MAC and ALU stages is only clocked when required.

Branch prediction

The integer core uses both static and dynamic branch prediction. All branches are predicted where the target address is an immediate address, or fixed-offset PC-relative address.

The first level of branch prediction is dynamic, through a 128-entry *Branch Target Address Cache* (BTAC). If the PC of a branch matches an entry in the BTAC, the branch history and the target address are used to fetch the new instruction stream.

Dynamically predicted branches might be removed from the instruction stream, and might execute in zero cycles.

If the address mappings are changed, the BTAC must be flushed. Address mapping changes are the result of page table mapping changes, FCSE PID Register changes, or Context ID Register changes. A BTAC flush instruction is provided in the CP15 coprocessor.

Static branch prediction is used to handle branches not matched in the BTAC. The static predictor makes a prediction based on the direction of the branches.

Return stack

A three-entry return stack is included to accelerate returns from procedure calls. For each procedure call, the return address is pushed onto a hardware stack. When a procedure return is recognized, the address held in the return stack is popped, and is used by the prefetch unit as the predicted return address.

Note

See *Pipeline stages* on page 1-22 for details of the pipeline stages and instruction progression.

See Chapter 3 *Control Coprocessor CP15* for system coprocessor programming information.

1.3.2 Load Store Unit (LSU)

The *Load Store Unit* (LSU) manages all load and store operations. The load-store pipeline decouples loads and stores from the MAC and ALU pipelines.

When LDM and STM instructions are issued to the LSU pipeline, other instructions run concurrently, subject to the requirements of supporting precise exceptions.

1.3.3 Prefetch unit

The prefetch unit fetches instructions from the Instruction Cache, or from external memory and predicts the outcome of branches in the instruction stream. See Chapter 6 *Program Flow Prediction* for more details.

1.3.4 Memory system

The core provides a level-one memory system with the following features:

- separate instruction and data caches
- separate instruction and data RAMs
- 64-bit datapaths throughout the memory system
- physically indexed, physically tagged data cache
- virtually indexed, physically tagged instruction cache
- complete memory management
- support for four sizes of memory page
- export of memory attributes for second-level memory system.

The memory system is described in more detail in the following sections:

- *Instruction and data caches* on page 1-10

- *Cache power consumption reduction*
- *Memory Management Unit* on page 1-11.

Instruction and data caches

The MP11 CPU provides separate instruction and data caches. The caches have the following features:

- The instruction and data cache can be independently configured during synthesis to sizes between 16KB and 64KB.
- Both caches are 4-way set-associative.
- Cache replacement policy is round-robin.
- The cache line length is eight words.
- Cache lines are write-back. The data cache is write-back write allocate.
- Each cache can be disabled independently, using the system control coprocessor.
- Both data cache read misses and write misses are non-blocking with up to three outstanding data cache read misses and up to four outstanding data cache write misses being supported.
- Support is provided for streaming of sequential data from LDM and LDRD operations, and for sequential instruction fetches.
- On a cache-miss, critical word first filling of the cache is performed.
- For optimum area and performance, all of the cache RAMs, and the associated tag RAMs, are designed to be implemented using standard ASIC RAM compilers.

Cache power consumption reduction

To reduce power consumption, the number of full cache reads is reduced by taking advantage of the sequential nature of many cache operations. If a cache read is sequential to the previous cache read, and the read is within the same cache line, only the data RAM set that was previously read is accessed.

Store buffer

The MP11 CPU has a store buffer with four 64-bit slots. The store buffer is responsible for dealing with all write transfers coming from the integer core. The store buffer can merge write accesses to the same 64-bit slot. If writes are performed to a cachable region the store buffer, when draining, performs a data cache lookup and subsequently writes data into the cache in the case of hits, and requests missing data in the case of misses.

Because cache lookups are only performed for writes to cachable regions and full line allocations from the store buffer to the cache are possible, the overall L1 and L2 write traffic is reduced.

Memory Management Unit

The *Memory Management Unit* (MMU) has a single *Translation Lookaside Buffer* (TLB) for both instructions and data. The MMU includes a 4KB page mapping size to enable a smaller RAM and ROM footprint for embedded systems and operating systems such as WindowsCE that have many small mapped objects. The MP11 CPUs implement the *Fast Context Switch Extension* (FCSE) and high vectors extension that are required to run Microsoft WindowsCE. See *c13, FCSE PID Register* on page 3-64 for more details.

The MMU is responsible for protection checking, address translation, and memory attributes, some of which can be passed to an external level two memory system. The memory translations are cached in MicroTLBs for each of the instruction and data caches, with a single Main TLB backing the MicroTLBs.

The MMU has the following features:

- matching of Virtual Address and ASID
- checking of domain access permissions
- checking of memory attributes
- virtual-to-physical address translation
- support for four page (region) sizes
- mapping of accesses to cache, or external memory
- TLB loading for hardware and software.

Paging

Four page sizes are supported:

- 16MB super sections
- 1MB sections
- 64KB large pages
- 4KB small pages.

Domains

Sixteen access domains are supported.

TLB

A two-level TLB structure is implemented. Eight entries in the main TLB are lockable. Hardware TLB loading is supported, and is backwards compatible with previous versions of the ARM architecture.

ASIDs

TLB entries can be global, or can be associated with particular processes or applications using *Application Space IDentifiers* (ASIDs). ASIDs enable TLB entries to remain resident during context switches, avoiding the requirement of reloading them subsequently, and also enable task-aware debugging.

System control coprocessor

Cache operations are controlled through a dedicated coprocessor, CP15, integrated within the core. This coprocessor provides a standard mechanism for configuring the level one memory system, and also provides functions such as memory barrier instructions. See *System control* on page 1-16 and Chapter 7 *Level 1 Memory System* for more details.

1.4 Debug and programming support

MP11 CPU debug and programming support features are described in:

- *Debug*
- *Vector Floating-Point (VFP)* on page 1-14
- *System control* on page 1-16
- *Interrupt handling* on page 1-16.

1.4.1 Debug

The debug coprocessor, CP14, implements a full range of debug features described in Chapter 12 *Debug* and Chapter 13 *Debug Test Access Port*.

The core provides extensive support for real-time debug and performance profiling.

Debug is described in more detail in the following sections:

- *System performance monitoring*
- *Real-time debug facilities*
- *Debug environment* on page 1-14.

System performance monitoring

This is a group of counters that can be configured to gather statistics on the operation of the processor and memory system. See *c15, Performance Monitor Control Register (PMNC)* on page 3-68 for more details.

Real-time debug facilities

The ARM11 MPCore processor contains an EmbeddedICE-RT logic unit to provide real-time debug facilities. It has the following capabilities:

- up to six breakpoints
- thread-aware breakpoints
- up to two watchpoints
- *Debug Communications Channel* (DCC).

The EmbeddedICE-RT logic is connected directly to the core and monitors the internal address and data buses. You can access the EmbeddedICE-RT logic in one of two ways:

- executing CP14 instructions
- through a JTAG-style interface and associated TAP controller.

The EmbeddedICE-RT logic supports two modes of debug operation:

Halt mode On a debug event, such as a breakpoint or watchpoint, the core is stopped and forced into debug state. This enables the internal state of the core, and the external state of the system, to be examined independently from other system activity. When the debugging process has been completed, the core and system state is restored, and normal program execution resumed.

Monitor debug-mode

On a debug event, a debug exception is generated instead of entering debug state, as in halt mode. A debug monitor program is activated by the exception entry and it is then possible to debug the processor while enabling the execution of critical interrupt service routines. The debug monitor program communicates with the debug host over the DCC.

Debug environment

Several external hardware and software tools are available to enable real-time debugging using the EmbeddedICE-RT logic.

1.4.2 Vector Floating-Point (VFP)

The optional VFP coprocessor within the ARM11 MPCore processor supports floating point arithmetic. The VFP is implemented as a dedicated functional block, and is mapped as coprocessor numbers 10 and 11. Using the coprocessor access register, software can determine whether the VFP is present (see *c1, Coprocessor Access Control Register* on page 3-36 for more details).

The VFP implements the ARM VFPv2 floating point coprocessor instruction set. It supports single and double-precision arithmetic on vector-vector, vector-scalar, and scalar-scalar data sets. Vectors can consist of up to eight single-precision, or four double-precision elements.

The VFP has its own bank of 32 registers for single-precision operands, which can be used in pairs for double-precision operands. Loads and stores of VFP registers can operate in parallel with arithmetic operations.

The VFP supports a wide range of single and double precision operations, including ABS, NEG, COPY, MUL, MAC, DIV, and SQRT. Most of these are effectively executed in a single cycle. Table 1-1 shows the exceptions. These issue latencies also apply to individual elements in a vector operation.

Table 1-1 Double-precision VFP operations

Instruction types	Issue latency
DP MUL and MAC	2 cycles
SP DIV, SQRT	14 cycles
DP DIV, SQRT	28 cycles
All other instructions	1 cycle

See *VFP11 Vector Floating-point Coprocessor Technical Reference Manual* for more details.

IEEE754 compliance

The VFP supports all five floating point exceptions defined by IEEE754:

- invalid operation
- divide by zero
- overflow
- underflow
- inexact.

Trapping of these exceptions can be individually enabled or disabled. If disabled, the IEEE754-defined default results are returned. All rounding modes are supported, and basic single and basic double formats are used.

For full compliance, support code is required to handle arithmetic where operands or results are de-norms. This support code is normally installed on the Undefined instruction exception handler.

Flush-to-zero mode

A flush-to-zero mode is provided where a default treatment of de-norms is applied. Table 1-2 shows the default behavior in flush-to-zero mode.

Table 1-2 Flush-to-zero mode

Operation	Flush-to-zero
De-norm operand(s)	Treated as 0+ Inexact flag set
De-norm result	Returned as 0+ Inexact Flag set

Operations not supported

The following operations are not directly supported by the VFP:

- remainder
- binary (decimal) conversions
- direct comparisons between single and double-precision values.

These are normally implemented as C library functions.

1.4.3 System control

The control of the memory system and its associated functionality, and other system-wide control attributes are managed through a dedicated system control coprocessor, CP15. See Chapter 3 *Control Coprocessor CP15* for more details.

1.4.4 Interrupt handling

Interrupt handling in the ARM11 MPCore processor is compatible with previous ARM architectures, but has several additional features to improve interrupt performance at interrupt generation and at interrupt handling levels.

Exception processing enhancements

The ARMv6 architecture contains several enhancements to exception processing, to reduce interrupt handler entry and exit time:

SRS	Save return state to a specified stack frame.
RFE	Return from exception.
CPS	Directly modify the CPSR.

Interrupt generation enhancements

The Distributed Interrupt Controller prioritizes and routes interrupts from various sources and redirects them to MP11 CPUs. See Chapter 10 *MPCore Distributed Interrupt Controller*.

1.5 Power management

The MP11 CPUs include several micro-architectural features to reduce energy consumption:

- Accurate branch and return prediction, reducing the number of incorrect instruction fetch and decode operations.
- Use of physically tagged caches, which reduce the number of cache flushes and refills, to save energy in the system.
- The use of MicroTLBs reduces the power consumed in translation and protection lookups for each memory access.
- The caches use sequential access information to reduce the number of accesses to the Tag RAMs and to unmatched data RAMs.
- Extensive use of gated clocks and gates to disable inputs to unused functional blocks. Because of this, only the logic actively in use to perform a calculation consumes any dynamic power.

The MP11 CPUs support the following levels of power management:

Run mode This mode is the normal mode of operation in which all of the functionality of the MP11 CPU is available.

Standby mode

This mode disables most of the clocks of the device, while keeping the device powered up. This reduces the power drawn to the static leakage current, plus a tiny clock power overhead required to enable the device to wake up from the standby state. The transition from the standby mode to the run mode is caused by one of the following:

- an interrupt, either masked or unmasked
- a debug request, regardless of whether debug is enabled
- reset.

Dormant mode

This mode enables the MP11 CPU to be powered down, while leaving the state of the caches powered up and maintaining their state. The following are required for full implementation of dormant mode:

- modification of the RAMs to include an input clamp
- implementation of separate power domains.

Shutdown mode

This mode has the entire MP11 CPU powered down. All state, including cache state, must be saved externally. The part is returned to the run state by the assertion of reset. This state saving is performed with interrupts disabled, and finishes with a Data Synchronization Barrier operation. The MP11 CPU then communicates with the power controller that it is ready to be powered down.

Power management features are described in more detail in Chapter 11 *Clocking, Resets, and Power Management*.

1.6 Configurable options

Table 1-3 shows the ARM11 MPCore processor configurable options.

Table 1-3 Configurable options for the ARM11 MPCore processor

Feature	Range of options
MP11 CPUs	One to four
Instruction cache size per MP11 CPU	16KB, 32KB, or 64KB
Data cache size per MP11 CPU	16KB, 32KB, or 64KB
Master ports	One or two
Width of interrupt bus	0-224 by increments of 32 pins
VFP per MP11 CPU	Included or not
Wrappers for power off and dormant modes	Included or not

The default configuration of the ARM11 MPCore processor is a four MP11 CPU configuration. Table 1-4 shows the default configuration of the ARM11 MPCore processor.

Table 1-4 Default configuration for the ARM11 MPCore processor

Feature	Default value
MP11 CPUs	Four
Instruction cache size per MP11 CPU	32KB
Data cache size per MP11 CPU	32KB
Master ports	Two
Width of interrupt bus	32 pins
VFP per MP11 CPU	Included
Wrappers for power off and dormant modes	Not included

For further information on configuring your ARM11 MPCore processor see the *ARM11 MPCore Processor Configuration Manual*.

The MBIST solution you choose must be configured to match the chosen MPCore cache sizes. In addition, the form of the MBIST solution for the RAM blocks in the MPCore design is determined when the processor is implemented. For details, see Appendix C *MBIST Controller and Dispatch Unit* and *ARM11 MPCore processor Configuration Manual*.

1.7 Pipeline stages

Figure 1-2 shows:

- the two Fetch stages
- a Decode stage
- an Issue stage
- the four stages of the MP11 CPU integer execution pipeline.

These eight stages make up the MP11 CPU pipeline.

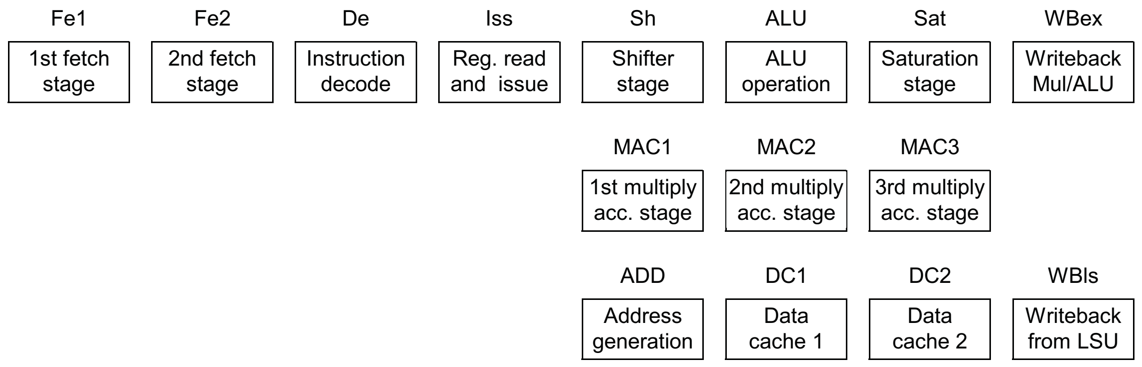


Figure 1-2 MP11 CPU pipeline stages

The pipeline stages are:

Fe1	First stage of instruction fetch and branch prediction.
Fe2	Second stage of instruction fetch and branch prediction.
De	Instruction decode.
Iss	Register read and instruction issue.
Sh	Shifter stage.
ALU	Main integer operation calculation.
Sat	Pipeline stage to enable saturation of integer results.
WBex	Write back of data from the multiply or main execution pipelines.
MAC1	First stage of the multiply-accumulate pipeline.
MAC2	Second stage of the multiply-accumulate pipeline.
MAC3	Third stage of the multiply-accumulate pipeline.
ADD	Address generation stage.
DC1	First stage of Data Cache access.
DC2	Second stage of Data Cache access.
WBls	Write back of data from the Load Store Unit.

By overlapping the various stages of operation, the MP11 CPU maximizes the clock rate achievable to execute each instruction. It delivers a throughput approaching one instruction for each cycle.

The Fetch stages can hold up to four instructions, where branch prediction is performed on instructions ahead of execution of earlier instructions.

The Issue and Decode stages can contain any instruction in parallel with a predicted branch.

The Execute, Memory, and Write stages can contain a predicted branch, an ALU or multiply instruction, a load/store multiple instruction, and a coprocessor instruction in parallel execution.

1.8 Typical pipeline operations

Figure 1-3 shows all the operations in each of the pipeline stages in the ALU pipeline, the load/store pipeline, and the HUM buffers.

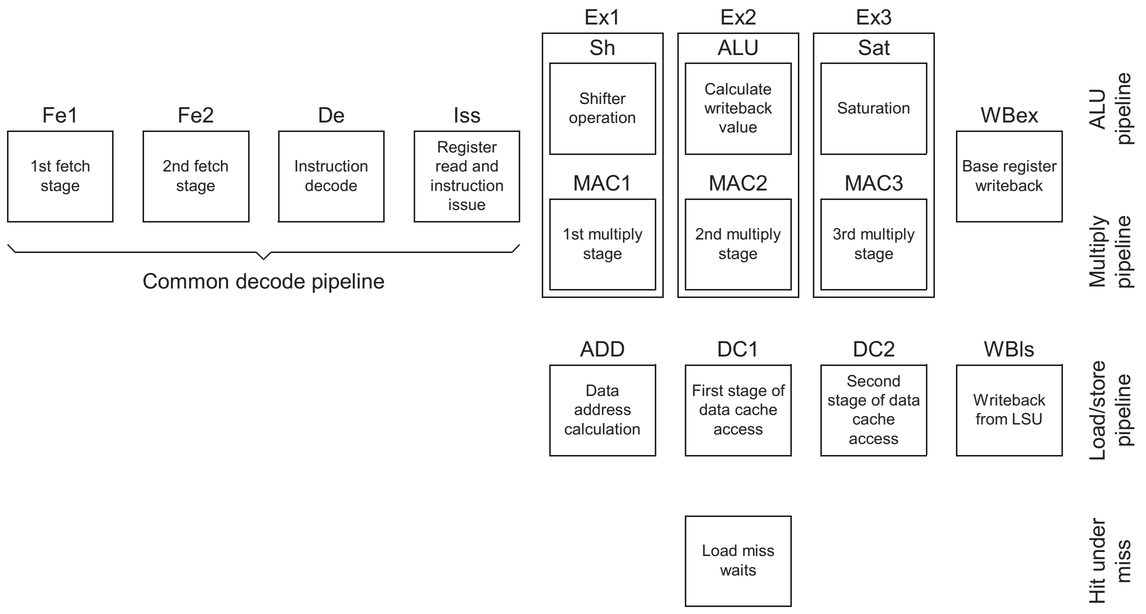


Figure 1-3 Typical operations in pipeline stages

Figure 1-4 shows a typical ALU data processing instruction. The load/store pipeline and the HUM buffer are not used.

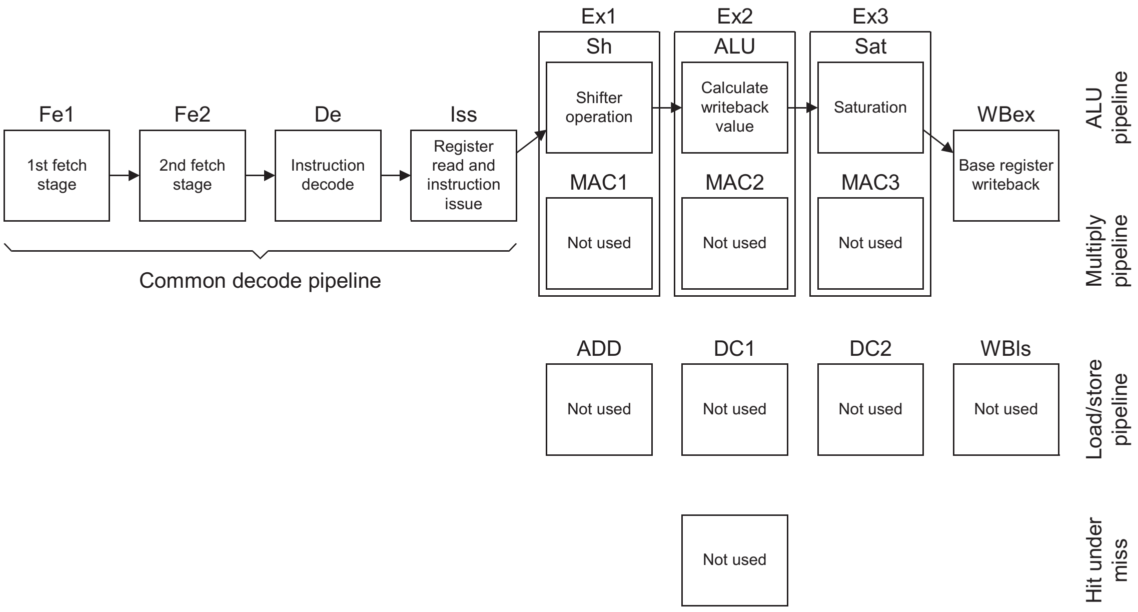


Figure 1-4 Typical ALU operation

Figure 1-5 shows a typical multiply operation. The MUL instruction can loop in the MAC1 stage until it has passed through the first part of the multiplier array enough times. Then it progresses to MAC2 and MAC3 where it passes once through the second half of the array to produce the final result.

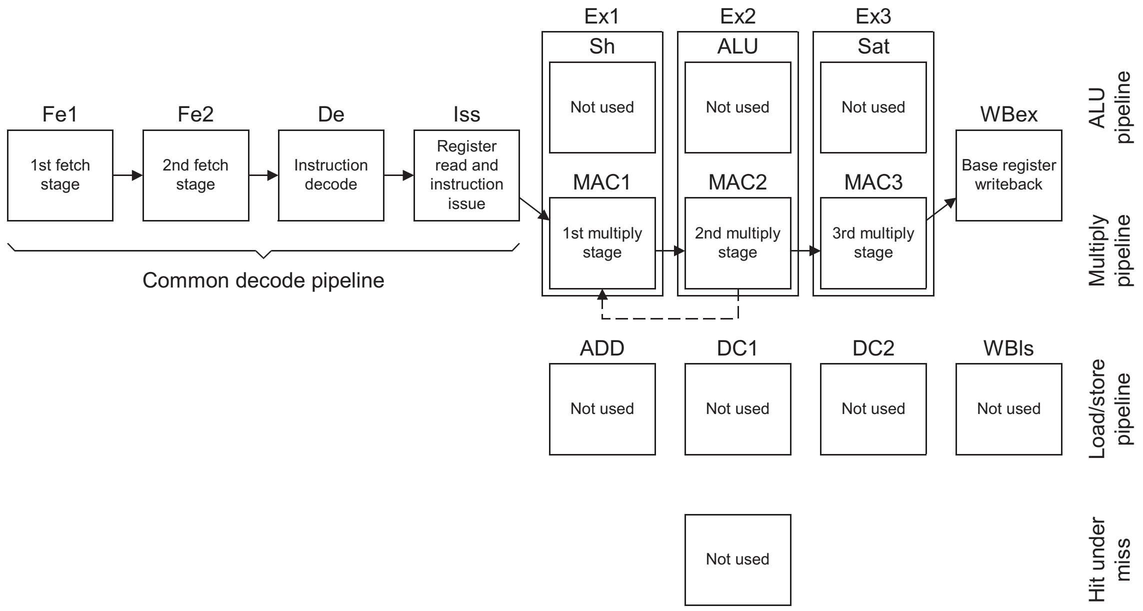


Figure 1-5 Typical multiply operation

1.8.1 Instruction progression

Figure 1-6 shows an LDR/STR operation that hits in the Data Cache.

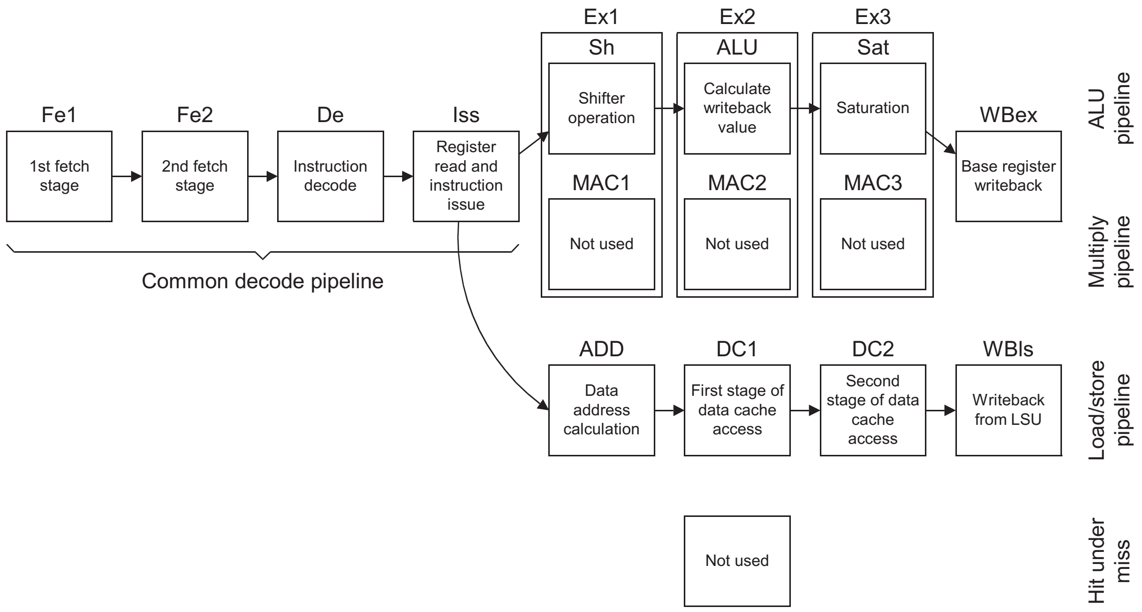


Figure 1-6 Progression of an LDR/STR operation

Figure 1-7 shows the progression of an LDM/STM operation using the load/store pipeline to complete. Other instructions can use the ALU pipeline at the same time as the LDM/STM completes in the load/store pipeline.

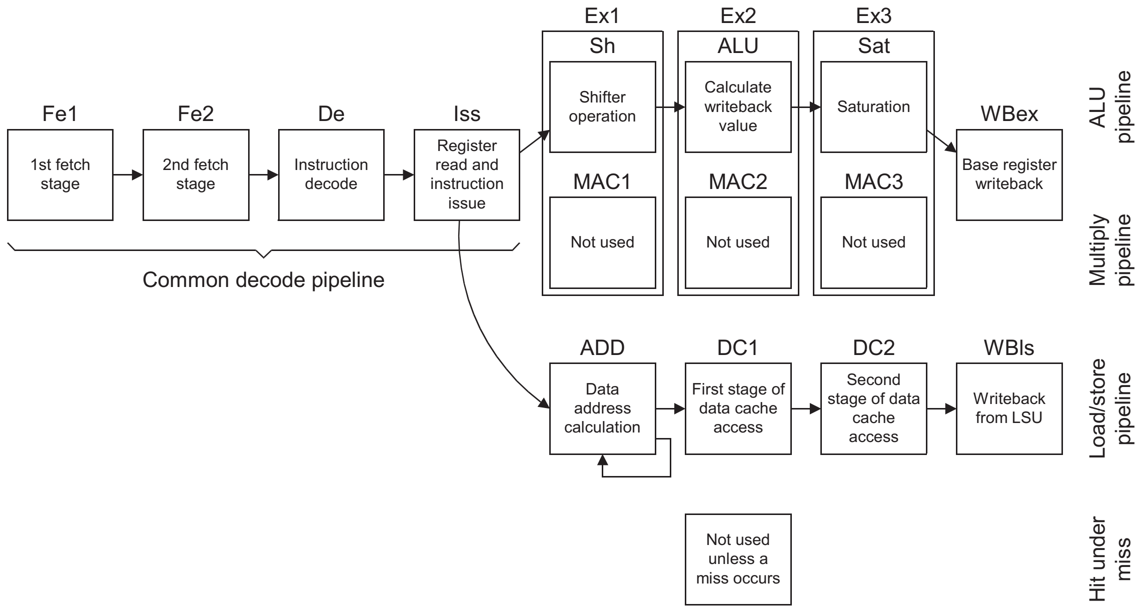


Figure 1-7 Progression of an LDM/STM operation

Figure 1-8 shows the progression of an LDR that misses. When the LDR is in the HUM buffers, other instructions, including independent loads that hit in the cache, can run under it.

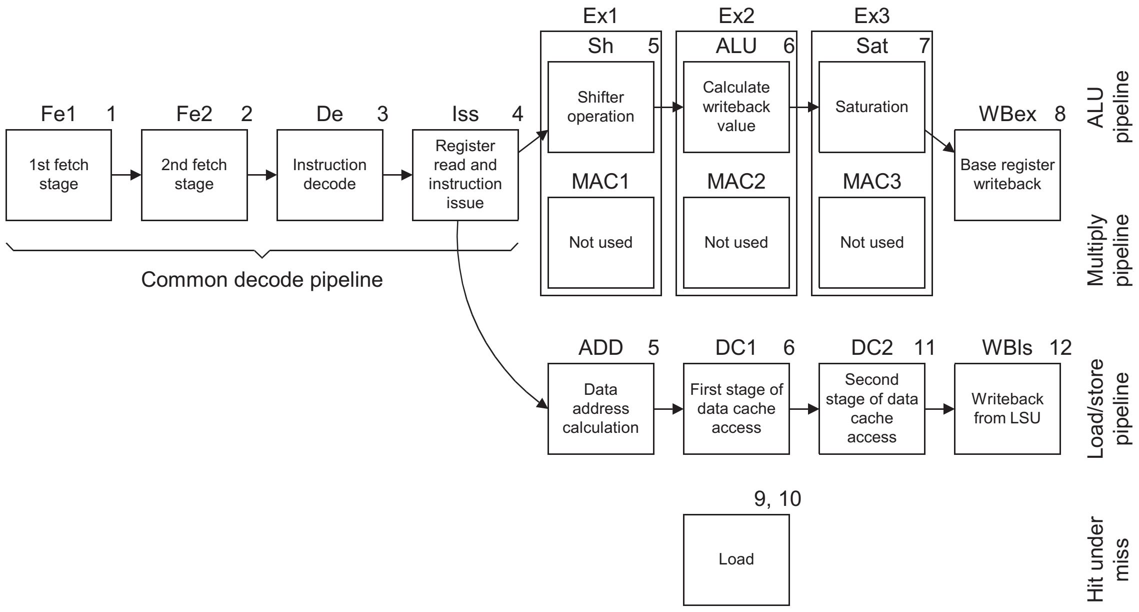


Figure 1-8 Progression of an LDR that misses

See Chapter 14 *Cycle Timings and Interlock Behavior* for details of instruction cycle timings.

1.9 MPCore architecture with Jazelle technology

The ARM11 MPCore processor has these instruction sets:

- the 32-bit ARM instruction set used in ARM state, with media instructions
- the 16-bit Thumb instruction set used in Thumb state
- the 8-bit Java bytecodes used in Java state.

For details of both the ARM and Thumb instruction sets, see the *ARM Architecture Reference Manual*. For full details of the MPCore Java instruction set, see the *Jazelle V1 Architecture Reference Manual*.

1.9.1 Instruction compression

A typical 32-bit architecture can manipulate 32-bit integers with single instructions, and address a large address space much more efficiently than a 16-bit architecture. When processing 32-bit data, a 16-bit architecture takes at least two instructions to perform the same task as a single 32-bit instruction.

When a 16-bit architecture has only 16-bit instructions, and a 32-bit architecture has only 32-bit instructions, overall the 16-bit architecture has higher code density, and greater than half the performance of the 32-bit architecture.

Thumb implements a 16-bit instruction set on a 32-bit architecture, giving higher performance than on a 16-bit architecture, with higher code density than a 32-bit architecture.

The ARM11 MPCore processor gives you the choice of running in ARM state, or Thumb state, or a mix of the two for each individual MP11 CPU. This enables you to optimize both code density and performance to best suit your application requirements.

1.9.2 The Thumb instruction set

The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions. Thumb instructions are 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect on the processor model. Thumb instructions operate with the standard ARM register configuration, enabling excellent interoperability between ARM and Thumb states.

Thumb has all the advantages of a 32-bit core:

- 32-bit address space
- 32-bit registers
- 32-bit shifter and *Arithmetic Logic Unit* (ALU)
- 32-bit memory transfer.

Thumb therefore offers a long branch range, powerful arithmetic operations, and a large address space.

The availability of both 16-bit Thumb and 32-bit ARM instruction sets, gives you the flexibility to emphasize performance or code size on a subroutine level, according to the requirements of their applications. For example, critical loops for applications such as fast interrupts and DSP algorithms can be coded using the full ARM instruction set, and linked with Thumb code.

1.9.3 Java bytecodes

ARM architecture v6 with Jazelle technology executes variable length Java bytecodes. Java bytecodes fall into two classes:

Hardware execution

Bytecodes that perform stack-based operations.

Software execution

Bytecodes that are too complex to execute directly in hardware are executed in software. An ARM register is used to access a table of exception handlers to handle these particular bytecodes.

A complete list of the MP11 CPU-supported Java bytecodes and their corresponding hardware or software instructions is in the *Jazelle V1 Architecture Reference Manual*.

1.10 Product revisions

This is the Technical Reference Manual for the ARM11 MPCore processor product revision r0p3. There is no change to the functionality described in this manual. See the engineering errata that accompanies the product deliverables for more information.

1.10.1 Additions and changes to the TRM

This section describes additions and changes to the text of the TRM:

- Table 3-40 on page 3-69 and Figure 3-41 on page 3-69, event counter names corrected.
- Table 8-1 on page 8-2, AXI master interface attributes added
- *Using the STRT instruction* on page 8-4 added.
- SCU CPU Register corrections, *SCU register definition* on page 9-3 and *SCU CPU Status Register* on page 9-6.

  Table 9-3 on page 9-5, clarification of bit 0 behavior.
- *Interrupt Distributor* on page 10-4, IPI priority clarified, receiver determines priority not sender.

  Interrupt configuration registers, 0xC00-0xC3C on page 10-16, information about ID29 and ID30 clarified. Information about ID31 added

  Table 10-9 on page 10-23, Watchdog Control Register value corrected. *Watchdog Control Register bit assignments* on page 10-27, for bit 3 the default is timer mode.
- *Clocking* on page 11-2, final sentence of the first paragraph rewritten for clarity.
- Figure 11-1 on page 11-12 redrawn.
- Figure E-4 on page E-5 redrawn.

Chapter 2
Programmer's Model

This chapter describes the MPCore registers and provides information for programming the microprocessor. It contains the following sections:

2.1 About the programmer's model

The ARM11 MPCore processor implements ARM architecture v6K with Java extensions. This includes the 32-bit ARM instruction set, 16-bit Thumb instruction set, and the 8-bit Java instruction set. For details of both the ARM and Thumb instruction sets, see the *ARM Architecture Reference Manual*. For the Java instruction set see the *Jazelle V1 Architecture Reference Manual*.

2.2 Processor operating states

The ARM11 MPCore processor has three operating states:

ARM state	32-bit, word-aligned ARM instructions are executed in this state.
Thumb state	16-bit, halfword-aligned Thumb instructions.
Java state	Variable length, byte-aligned Java instructions.

In Thumb state, the *Program Counter* (PC) uses bit 1 to select between alternate halfwords. In Java state, all instruction fetches are in words.

Note

Transition between ARM and Thumb states does not affect the processor mode or the register contents. For details on entering and exiting Java state see *Jazelle V1 Architecture Reference Manual*.

2.2.1 Switching state

You can switch the operating state of the ARM11 MPCore processor between:

- ARM state and Thumb state using the BX and BLX instructions, and loads to the PC. Switching state is described in the *ARM Architecture Reference Manual*.
- ARM state and Java state using the BXJ instruction.

All exceptions are entered, handled, and exited in ARM state. If an exception occurs in Thumb state or Java state, the processor reverts to ARM state. Exception return instructions restore the SPSR to the CPSR, which can also cause a transition back to Thumb state or Java state.

2.2.2 Interworking ARM and Thumb state

The ARM11 MPCore processor enables you to mix ARM and Thumb code. For details see the chapter about interworking ARM and Thumb in the *RealView Compilation Tools Developer Guide*.

2.3 Instruction length

Instructions are one of:

- 32 bits long (in ARM state)
- 16 bits long (in Thumb state)
- variable length, multiples of 8 bits (in Java state).

2.4 Data types

The ARM11 MPCore processor supports the following data types:

- word (32-bit)
- halfword (16-bit)
- byte (8-bit).

In addition, long multiplies support 64-bit results with or without accumulation.

Note

- When any of these types are described as unsigned, the N-bit data value represents a non-negative integer in the range 0 to $+2^{N}-1$, using normal binary format.
- When any of these types are described as signed, the N-bit data value represents an integer in the range -2^{N-1} to $+2^{N-1}-1$, using two's complement format.
- Where the results of signed and unsigned versions of an instruction differ, both versions are usually provided. The main exception is that both versions are provided for only some of the multiply and multiply-accumulate instructions.

For best performance you must align these as follows:

- word quantities must be aligned to four-byte boundaries
- halfword quantities must be aligned to two-byte boundaries
- byte quantities can be placed on any byte boundary.

Note

You cannot use LDRD, LDM, LDC, STRD, STM, or STC instructions to access 32-bit quantities if they are unaligned.

2.5 Memory formats

The ARM11 MPCore processor views memory as a linear collection of bytes numbered in ascending order from zero. Bytes 0-3 hold the first stored word, and bytes 4-7 hold the second stored word, for example.

The ARM11 MPCore processor can treat words in memory as being stored in either:

- *32-bit byte-invariant BE-8 format*
- *Little-endian format*.

2.5.1 32-bit byte-invariant BE-8 format

In a byte-invariant system, the address of each byte of memory remains unchanged when switching between little-endian and big-endian operation. When a data item larger than a byte is loaded from or stored to memory, the bytes making up that data item are arranged into the correct order depending on the endianness of the memory access. Figure 2-1 shows the most significant byte of words at the lowest-numbered bytes.

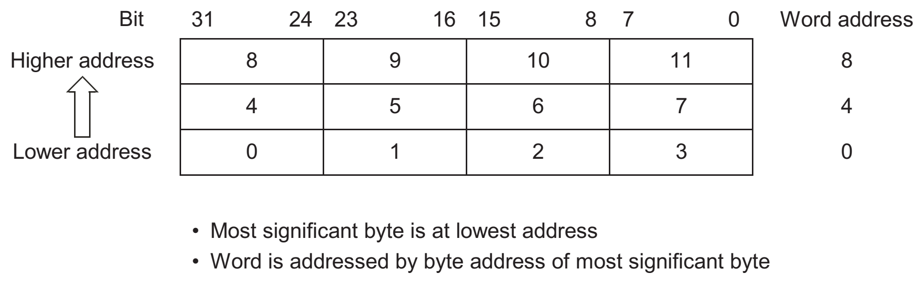
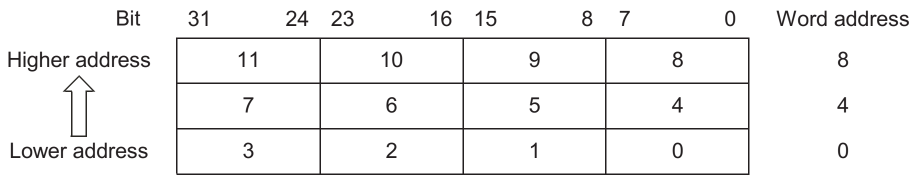

Bit
31
24
23
16
15
8
7
0
Word address
Higher address
Lower address
8
9
10
11
8
4
5
6
7
4
0
1
2
3
0
• Most significant byte is at lowest address
• Word is addressed by byte address of most significant byte

Figure 2-1 Big-endian addresses of bytes within words

2.5.2 Little-endian format

In little-endian format, the lowest-numbered byte in a word is the least significant byte of the word and the highest-numbered byte is the most significant. Therefore, byte 0 of the memory system connects to data lines 7-0. Figure 2-2 on page 2-7 shows this.



- Least significant byte is at lowest address
- Word is addressed by byte address of least significant byte

Figure 2-2 Little-endian addresses of bytes within words

2.6 Addresses in an MPCore system

Three distinct types of address exist in an MPCore system:

- *Virtual Address* (VA)
- *Modified Virtual Address* (MVA)
- *Physical Address* (PA).

Table 2-1 shows the address types in an MPCore system.

Table 2-1 Address types in an MPCore system

MP11 CPU integer core	Caches		TLBs	AMBA bus
	Instruction	Data		
Virtual Address	Virtual index physical tag	Physical index physical tag	Translates Virtual Address to Physical Address	Physical Address

This is an example of the address manipulation that occurs when an MP11 CPU requests an instruction:

1. The VA of the instruction is issued by the MP11 CPU integer core.
2. The Instruction Cache is indexed by the lower bits of the VA. The VA is translated using the ProcID to the MVA, and then to PA in the *Translation Lookaside Buffer* (TLB). The TLB performs the translation in parallel with the cache lookup.
3. If the protection check carried out by the TLB on the MVA does not abort and the PA tag is in the Instruction Cache, the instruction data is returned to the MP11 CPU integer core.
4. The PA is passed to the AMBA bus interface to perform an external access, in the event of a cache miss.

This is an example of the address manipulation that occurs when the MP11 CPU performs a data read.

1. The VA of the data access is issued by the MP11 CPU integer core.
2. The VA is translated using the ProcID to the MVA, and then to PA in the *Translation Lookaside Buffer* (TLB).
3. If the protection check carried out by the TLB on the MVA does not abort. the data access is processed in the level 1 memory stem based on its memory attributes retrieved in step 2.

4. If an external access is required, the Physical Address retrieved in step 2 is used to perform an access on the AMBA bus.

2.7 Operating modes

In all states there are seven modes of operation:

- User mode is the usual ARM program execution state, and is used for executing most application programs
- *Fast interrupt* (FIQ) mode is used for handling fast interrupts
- *Interrupt* (IRQ) mode is used for general-purpose interrupt handling
- Supervisor mode is a protected mode for the operating system
- Abort mode is entered after a data or instruction Prefetch Abort
- System mode is a privileged user mode for the operating system
- Undefined mode is entered when an Undefined instruction exception occurs.

Modes other than User mode are collectively known as privileged modes. Privileged modes are used to service interrupts or exceptions, or to access protected resources.

2.8 Registers

The ARM11 MPCore processor has a total of 37 registers:

- 31 general-purpose 32-bit registers
- six 32-bit status registers.

These registers are not all accessible at the same time. The processor state and operating mode determine which registers are available to the programmer.

2.8.1 The ARM state register set

In ARM state, 16 general registers and one or two status registers are accessible at any time. In privileged modes, mode-specific banked registers become available. Figure 2-3 on page 2-13 shows which registers are available in each mode.

The ARM state register set contains 16 directly-accessible registers, r0-r15. Another register, the *Current Program Status Register* (CPSR), contains condition code flags, status bits, and current mode bits. Registers r0-r13 are general-purpose registers used to hold either data or address values. Registers r14, r15, and the SPSR have the following special functions:

Link Register Register r14 is used as the subroutine *Link Register* (LR).

Register r14 receives the return address when a *Branch with Link* (BL or BLX) instruction is executed.

You can treat r14 as a general-purpose register at all other times. The corresponding banked registers r14_svc, r14_irq, r14_fiq, r14_abt, and r14_und are similarly used to hold the return values when interrupts and exceptions arise, or when BL or BLX instructions are executed within interrupt or exception routines.

Program Counter Register r15 holds the PC:

- in ARM state this is word-aligned
- in Thumb state this is halfword-aligned
- in Java state this is byte-aligned.

Saved Program Status Register

In privileged modes, another register, the *Saved Program Status Register* (SPSR), is accessible. This contains the condition code flags, status bits, and current mode bits saved as a result of the exception that caused entry to the current mode.

Banked registers have a mode identifier that indicates which mode they relate to. These mode identifiers are listed in Table 2-2.

Table 2-2 Register mode identifiers

Mode	Mode identifier
User	usr[a]
Fast interrupt	fiq
Interrupt	irq
Supervisor	svc
Abort	abt
System	usr[a]
Undefined	und

a. The `usr` identifier is usually omitted from register names. It is only used in descriptions where the User or System mode register is specifically accessed from another operating mode.

FIQ mode has seven banked registers mapped to r8–r14 (r8_fiq–r14_fiq). As a result many FIQ handlers do not have to save any registers.

The Supervisor, Abort, IRQ, and Undefined modes each have alternative mode-specific registers mapped to r13 and r14, permitting a private stack pointer and link register for each mode.

Figure 2-3 on page 2-13 shows the ARM state registers.

ARM state general registers and program counter

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	◺ r8_fiq	r8	r8	r8	r8
r9	◺ r9_fiq	r9	r9	r9	r9
r10	◺ r10_fiq	r10	r10	r10	r10
r11	◺ r11_fiq	r11	r11	r11	r11
r12	◺ r12_fiq	r12	r12	r12	r12
r13	◺ r13_fiq	◺ r13_svc	◺ r13_abt	◺ r13_irq	◺ r13_und
r14	◺ r14_fiq	◺ r14_svc	◺ r14_abt	◺ r14_irq	◺ r14_und
r15	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

ARM state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	◺ SPSR_fiq	◺ SPSR_svc	◺ SPSR_abt	◺ SPSR_irq	◺ SPSR_und

◺ = banked register

Figure 2-3 Register organization in ARM state

Figure 2-4 on page 2-14 shows an alternative view of the ARM registers.

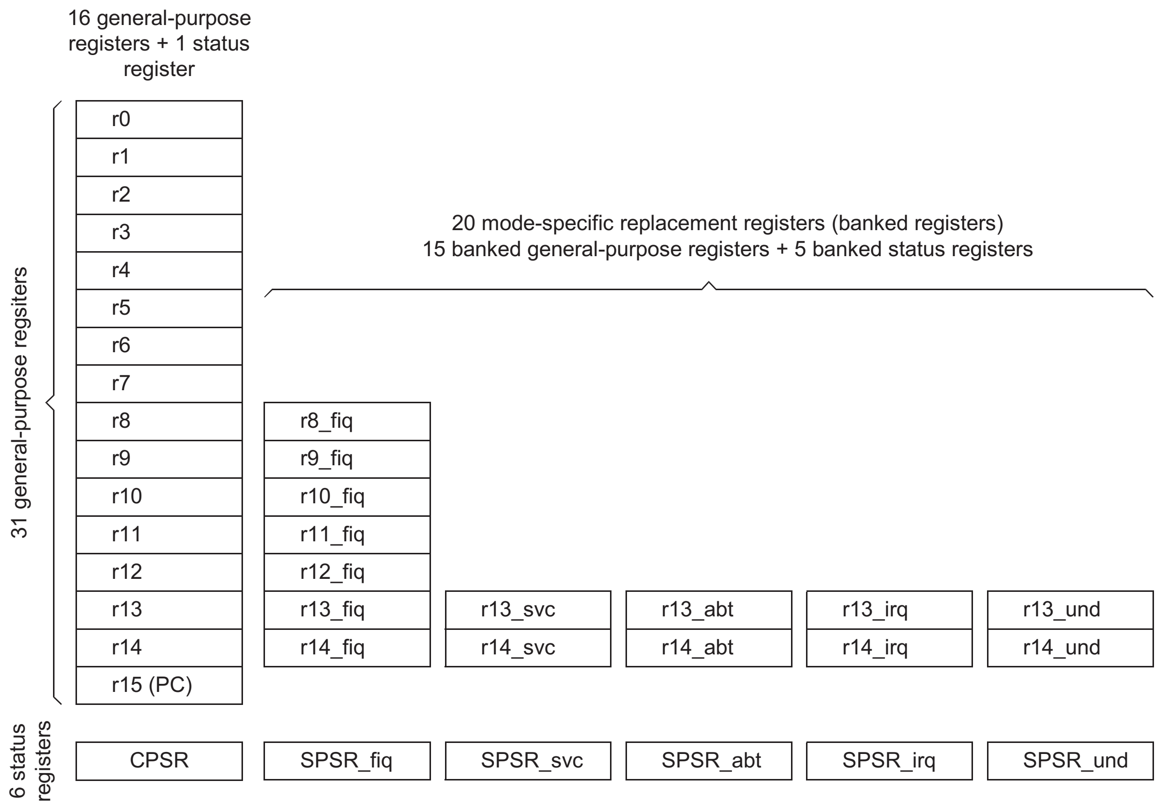


Figure 2-4 MPCore register set showing banked registers

2.8.2 The Thumb state register set

The Thumb state register set is a subset of the ARM state set. The programmer has direct access to:

- eight general registers, r0–r7 (for details of high register access in Thumb state see *Accessing high registers in Thumb state* on page 2-15)
- the PC
- a stack pointer, SP (ARM r13)
- an LR (ARM r14)
- the CPSR.

There are banked SPs, LRs, and SPSRs for each privileged mode. Figure 2-5 on page 2-15 shows the organization of Thumb state register set.

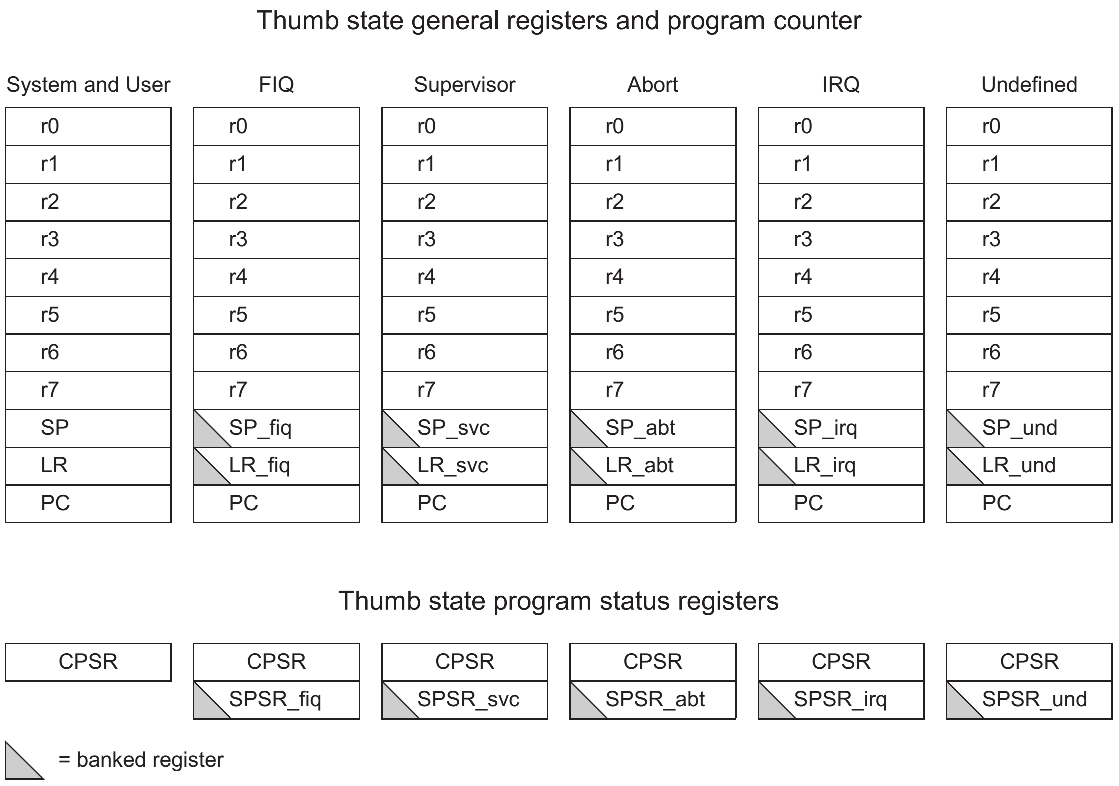


Figure 2-5 Register organization in Thumb state

2.8.3 Accessing high registers in Thumb state

In Thumb state, the high registers, r8–r15, are not part of the standard register set. You can use special variants of the MOV instruction to transfer a value from a low register, in the range r0–r7, to a high register, and from a high register to a low register. The CMP instruction enables you to compare high register values with low register values. The ADD instruction enables you to add high register values to low register values. For more details, see the *ARM Architecture Reference Manual*.

2.8.4 ARM state and Thumb state registers relationship

Figure 2-6 on page 2-16 shows the relationships between the Thumb state and ARM state registers. See the *Jazelle V1 Architecture Reference Manual* for details of Java state registers.

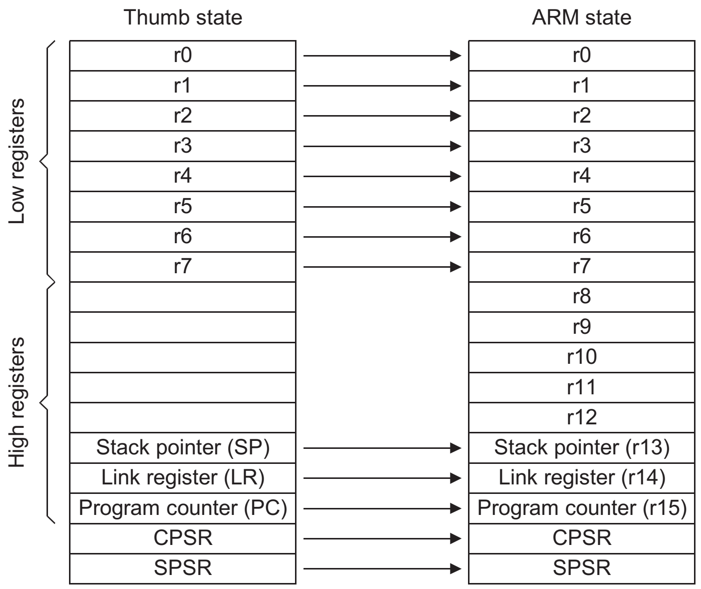


Figure 2-6 ARM state and Thumb state registers relationship

Note

Registers r0–r7 are known as the low registers. Registers r8–r15 are known as the high registers.

2.9 The program status registers

The ARM11 MPCore processor contains one CPSR, and five SPSRs for exception handlers to use. The program status registers:

- hold information about the most recently performed ALU operation
- control the enabling and disabling of interrupts
- set the processor operating mode.

Figure 2-7 shows the arrangement of bits in the status registers. These are described in the sections from *The condition code flags* to *Reserved bits* on page 2-24 inclusive.

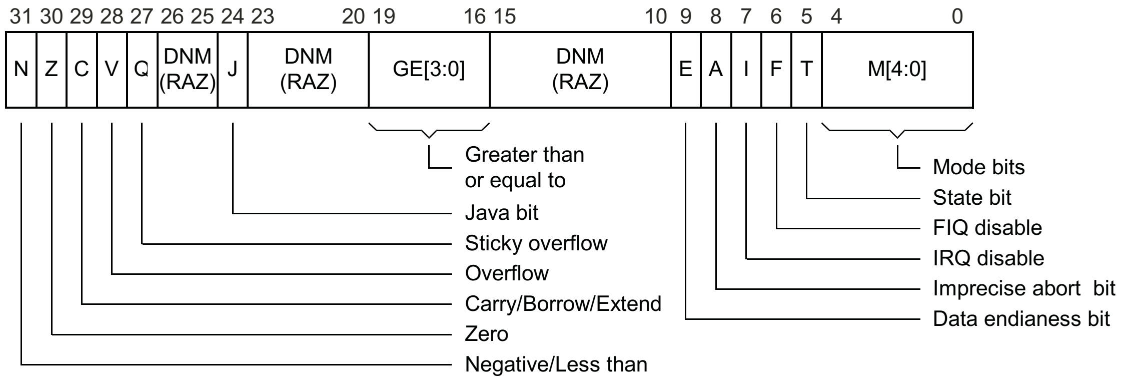


Figure 2-7 Program status register

Note

The bits identified in Figure 2-7 as *Do Not Modify* (DNM) (*Read As Zero* (RAZ)) must not be modified by software. These bits are:

- Readable, to enable the processor state to be preserved (for example, during process context switches)
- Writable, to enable the processor state to be restored. To maintain compatibility with future ARM processors, and as good practice, you are strongly advised to use a read-modify-write strategy when changing the CPSR.

2.9.1 The condition code flags

The N, Z, C, and V bits are the condition code flags. You can set them by arithmetic and logical operations, and also by MSR and LDM instructions. The ARM11 MPCore processor tests these flags to determine whether to execute an instruction.

In ARM state, most instructions can execute conditionally on the state of the N, Z, C, and V bits. The exceptions are:

- BKPT
- CDP2
- CPS
- LDC2
- MCR2
- MCRR2
- MRC2
- MRRC2
- PLD
- SETEND
- RFE
- SRS
- STC2.

In Thumb state, only the Branch instruction can be executed conditionally. For more details about conditional execution, see the *ARM Architecture Reference Manual*.

2.9.2 The Q flag

The Sticky Overflow (Q) flag can be set by certain multiply and fractional arithmetic instructions:

- QADD
- QDADD
- QSUB
- QDSUB
- SMLAD
- SMLAxy
- SMLAWy
- SMLSD
- SMUAD
- SSAT
- SSAT16
- USAT
- USAT16.

The Q flag is sticky in that, when set by an instruction, it remains set until explicitly cleared by an MSR instruction writing to the CPSR. Instructions cannot execute conditionally on the status of the Q flag.

To determine the status of the Q flag you must read the PSR into a register and extract the Q flag from this. For details of how the Q flag is set and cleared, see individual instruction definitions in the *ARM Architecture Reference Manual*.

2.9.3 The J bit

The J bit in the CPSR indicates when an MP11 CPU is in Java state.

When:

J = 0	The processor is in ARM or Thumb state, depending on the T bit.
J = 1	The processor is in Java state.

Note

- The combination of J = 1 and T = 1 causes similar effects to setting T=1 on a non Thumb-aware processor. That is, the next instruction executed causes entry to the Undefined Instruction exception. Entry to the exception handler causes the processor to re-enter ARM state, and the handler can detect that this was the cause of the exception because J and T are both set in SPSR_und.
- MSR cannot be used to change the J bit in the CPSR.
- The placement of the J bit avoids the status or extension bytes in code running on ARMv5TE or earlier processors. This ensures that OS code written using the deprecated CPSR, SPSR, CPSR_all, or SPSR_all syntax for the destination of an MSR instruction continues to work.

2.9.4 The GE[3:0] bits

Some of the SIMD instructions set GE[3:0] as greater-than-or-equal bits for individual halfwords or bytes of the result. Table 2-3 on page 2-20 shows this.

Table 2-3 GE[3:0] settings

Instruction	GE[3] A op B > C	GE[2] A op B > C	GE[1] A op B > C	GE[0] A op B > C
Signed				
SADD16	[31:16] + [31:16] $\geq$ 0	[31:16] + [31:16] $\geq$ 0	[15:0] + [15:0] $\geq$ 0	[15:0] + [15:0] $\geq$ 0
SSUB16	[31:16] - [31:16] $\geq$ 0	[31:16] - [31:16] $\geq$ 0	[15:0] - [15:0] $\geq$ 0	[15:0] - [15:0] $\geq$ 0
SADDSUBX	[31:16] + [15:0] $\geq$ 0	[31:16] + [15:0] $\geq$ 0	[15:0] - [31:16] $\geq$ 0	[15:0] - [31:16] $\geq$ 0
SSUBADDX	[31:16] - [15:0] $\geq$ 0	[31:16] - [15:0] $\geq$ 0	[15:0] + [31:16] $\geq$ 0	[15:0] + [31:16] $\geq$ 0
SADD8	[31:24] + [31:24] $\geq$ 0	[23:16] + [23:16] $\geq$ 0	[15:8] + [15:8] $\geq$ 0	[7:0] + [7:0] $\geq$ 0
SSUB8	[31:24] - [31:24] $\geq$ 0	[23:16] - [23:16] $\geq$ 0	[15:8] - [15:8] $\geq$ 0	[7:0] - [7:0] $\geq$ 0
Unsigned				
UADD16	[31:16] + [31:16] $\geq 2^{16}$	[31:16] + [31:16] $\geq 2^{16}$	[15:0] + [15:0] $\geq 2^{16}$	[15:0] + [15:0] $\geq 2^{16}$
USUB16	[31:16] - [31:16] $\geq$ 0	[31:16] - [31:16] $\geq$ 0	[15:0] - [15:0] $\geq$ 0	[15:0] - [15:0] $\geq$ 0
UADDSUBX	[31:16] + [15:0] $\geq 2^{16}$	[31:16] + [15:0] $\geq 2^{16}$	[15:0] - [31:16] $\geq$ 0	[15:0] - [31:16] $\geq$ 0
USUBADDX	[31:16] - [15:0] $\geq$ 0	[31:16] - [15:0] $\geq$ 0	[15:0] + [31:16] $\geq 2^{16}$	[15:0] + [31:16] $\geq 2^{16}$
UADD8	[31:24] + [31:24] $\geq 2^{8}$	[23:16] + [23:16] $\geq 2^{8}$	[15:8] + [15:8] $\geq 2^{8}$	[7:0] + [7:0] $\geq 2^{8}$
USUB8	[31:24] - [31:24] $\geq$ 0	[23:16] - [23:16] $\geq$ 0	[15:8] - [15:8] $\geq$ 0	[7:0] - [7:0] $\geq$ 0

Note

GE bit is 1 if A op B $\geq$ C, otherwise 0.

The SEL instruction uses GE[3:0] to select which source register supplies each byte of its result.

Note

- For unsigned operations, the GE bits are determined by the usual ARM rules for carries out of unsigned additions and subtractions, and so are carry-out bits.

- For signed operations, the rules for setting the GE bits are chosen so that they have the same sort of greater than or equal functionality as for unsigned operations.

2.9.5 The E bit

ARM and Thumb instructions are provided to set and clear the E-bit. The E bit controls load/store endianness.

Architecture versions prior to ARMv6 specify this bit as SBZ. This ensures no endianness reversal on loads or stores.

2.9.6 The A bit

The A bit is set automatically. It is used to disable imprecise Data Aborts. For details of how to use the A bit see *Imprecise Data Abort mask in the CPSR/SPSR* on page 2-31.

2.9.7 The control bits

The bottom eight bits of a PSR are known collectively as the control bits. They are the:

- *Interrupt disable bits*
- *T bit* on page 2-22
- *Mode bits* on page 2-22.

The control bits change when an exception occurs. When the processor is operating in a privileged mode, software can manipulate these bits.

Interrupt disable bits

The I and F bits are the interrupt disable bits:

- when the I bit is set, IRQ interrupts are disabled
- when the F bit is set, FIQ interrupts are disabled.

MP11 CPUs also have *Non Maskable Fast Interrupt* (NMFI). NMFI behavior is as follows:

- only **FIQISNMI** controls this behavior. There is no software control.
- Values can be read from bit 27 in CP15 register 1, the NMFI bit.

  1 = FIQs behave as NMFIs

  0 = Normal FIQ behavior.

- All instruction-generated writes of a value to the CPSR F-bit are changed to AND the new value into the F bit. In other words, writes of 0 clear the F bit and writes of 1 leave it unchanged.
- The CPSR F-bit is set only by taking an FIQ exception. Other methods of entering FIQ mode, such as using MSR to write 0b10001 to the CPSR mode bits, do not cause the F bit to become set nor allow instructions to set it.

T bit

The T bit reflects the operating state:

- when the T bit is set, the processor is executing in Thumb state
- when the T bit is clear, the processor is executing in ARM state, or Java state depending on the J bit.

Note

Never use an MSR instruction to force a change to the state of the T bit in the CPSR. If an MSR instruction does try to modify this bit the result is architecturally Unpredictable. In the ARM11 MPCore processor writes to this bit are ignored.

Mode bits

Caution

An illegal value programmed into M[4:0] causes the processor to enter an unrecoverable state. If this occurs, you must apply reset. Not all combinations of the mode bits define a valid processor mode, so take care to use only those bit combinations shown.

Table 2-4 shows the M[4:0] mode bits that are used to determine the processor operating mode.

Table 2-4 PSR mode bit values

M[4:0]	Mode	Visible state registers	
		Thumb	ARM
b10000	User	r0–r7, r8-r12[a], SP, LR, PC, CPSR	r0–r14, PC, CPSR
b10001	FIQ	r0–r7, r8_fiq-r12_fiq[a], SP_fiq, LR_fiq PC, CPSR, SPSR_fiq	r0–r7, r8_fiq–r14_fiq, PC, CPSR, SPSR_fiq

Table 2-4 PSR mode bit values (continued)

M[4:0]	Mode	Visible state registers	
		Thumb	ARM
b10010	IRQ	r0–r7, r8-r12[a], SP_irq, LR_irq, PC, CPSR, SPSR_irq	r0–r12, r13_irq, r14_irq, PC, CPSR, SPSR_irq
b10011	Supervisor	r0–r7, r8-r12[a], SP_svc, LR_svc, PC, CPSR, SPSR_svc	r0–r12, r13_svc, r14_svc, PC, CPSR, SPSR_svc
b10111	Abort	r0–r7, r8-r12[a], SP_abt, LR_abt, PC, CPSR, SPSR_abt	r0–r12, r13_abt, r14_abt, PC, CPSR, SPSR_abt
b11011	Undefined	r0–r7, r8-r12[a], SP_und, LR_und, PC, CPSR, SPSR_und	r0–r12, r13_und, r14_und, PC, CPSR, SPSR_und
b11111	System	r0–r7, r8-r12[a], SP, LR, PC, CPSR	r0–r14, PC, CPSR

a. Access to these registers is limited in Thumb state.

2.9.8 Modification of PSR bits by MSR instructions

In previous architecture versions, MSR instructions can modify the flags byte, bits [31:24], of the CPSR in any mode, but the other three bytes are only modifiable in privileged modes.

After the introduction of ARM architecture v6, however, each CPSR bit falls into one of the following categories:

- Bits that are freely modifiable from any mode, either directly by MSR instructions or by other instructions whose side-effects include writing the specific bit or writing the entire CPSR.

  Bits in Figure 2-7 on page 2-17 that are in this category are N, Z, C, V, Q, GE[3:0], and E.

- Bits that must never be modified by an MSR instruction, and so must only be written as a side-effect of another instruction. If an MSR instruction does try to modify these bits the results are architecturally Unpredictable. In the ARM11 MPCore processor these bits are not affected.

  Bits in Figure 2-7 on page 2-17 that are in this category are J and T.

- Bits that can only be modified from privileged modes, and that are completely protected from modification by instructions while the processor is in User mode. The only way that these bits can be modified while the processor is in User mode is by entering a processor exception, as described in *Exceptions* on page 2-25.

Bits in Figure 2-7 on page 2-17 that are in this category are A, I, F, and M[4:0].

2.9.9 Reserved bits

The remaining bits in the PSRs are unused, but are reserved. When changing a PSR flag or control bits, make sure that these reserved bits are not altered. You must ensure that your program does not rely on reserved bits containing specific values because future processors might use some or all of the reserved bits.

2.10 Exceptions

Exceptions occur whenever the normal flow of a program must be halted temporarily. For example, to service an interrupt from a peripheral. Before attempting to handle an exception, the MP11 CPU preserves the current processor state so that the original program can resume when the handler routine has finished.

If two or more exceptions occur simultaneously, the exceptions are dealt with in the fixed order given in *Exception priorities* on page 2-34.

This section provides details of the MPCore exception handling:

- *Exception entry and exit summary* on page 2-26
- *Entering an ARM exception* on page 2-27
- *Leaving an ARM exception* on page 2-28.

Several enhancements are made in ARM architecture v6 to the exception model, mostly to improve interrupt latency, as follows:

- New instructions are added to give a choice of stack to use for storing the exception return state after exception entry, and to simplify changes of processor mode and the disabling and enabling of interrupts.
- Support for an imprecise Data Abort that behaves as an interrupt rather than as an abort, in that it occurs asynchronously relative to the instruction execution. Support involves the masking of a pending imprecise Data Abort at times when entry into Abort mode is deemed unrecoverable.

2.10.1 New instructions for exception handling

This section describes the instructions added to accelerate the handling of exceptions. Full details of these instructions are given in the *ARM Architecture Reference Manual*.

Store Return State (SRS)

This instruction stores r14_<current_mode> and spsr_<current_mode> to sequential addresses, using the banked version of r13 for a specified mode to supply the base address (and to be written back to if base register write-back is specified). This enables an exception handler to store its return state on a stack other than the one automatically selected by its exception entry sequence.

The addressing mode used is a version of ARM addressing mode 4 (see the *ARM Architecture Reference Manual*, *Part A*), modified to assume a {r14,SPSR} register list, rather than using a list specified by a bit mask in the instruction. This enables the SRS instruction to access stacks in a manner compatible with the normal use of STM instructions for stack accesses.

Return From Exception (RFE)

This instruction loads the PC and CPSR from sequential addresses. This is used to return from an exception that has had its return state saved using the SRS instruction (see *Store Return State (SRS)* on page 2-25), and again uses a version of ARM addressing mode 4, modified this time to assume a {PC, CPSR} register list.

Change Processor State (CPS)

This instruction provides new values for the CPSR interrupt masks, mode bits, or both, and is designed to shorten and speed up the read/modify/write instruction sequence used in ARMv5 to perform such tasks. Together with the SRS instruction, it enables an exception handler to save its return information on the stack of another mode and then switch to that other mode, without modifying the stack belonging to the original mode or any registers other than the new mode stack pointer.

This instruction also streamlines interrupt mask handling and mode switches in other code. In particular it enables short code sequences to be made atomic efficiently in a uniprocessor system by disabling interrupts at their start and re-enabling interrupts at their end. A similar Thumb instruction is also provided. However, the Thumb instruction can only change the interrupt masks, not the processor mode as well, to avoid using too much instruction set space.

2.10.2 Exception entry and exit summary

Table 2-5 summarizes the PC value preserved in the relevant r14 on exception entry, and the recommended instruction for exiting the exception handler. Full details of Java state exceptions are provided in the *Jazelle V1 Architecture Reference Manual*.

Table 2-5 Exception entry and exit

Exception on entry	Return instruction	Previous state			Notes
		ARM r14_x	Thumb r14_x	Java r14_x	
SWI	`MOVS PC, R14_svc`	PC + 4	PC+2	-	Where the PC is the address of the SWI or Undefined instruction. Not used in Java state.
UNDEF	`MOVS PC, R14_und`	PC + 4	PC+2	-	
PABT	`SUBS PC, R14_abt, #4`	PC + 4	PC+4	PC+4	Where the PC is the address of instruction that had the Prefetch Abort.

Table 2-5 Exception entry and exit (continued)

Exception on entry	Return instruction	Previous state			Notes
		ARM r14_x	Thumb r14_x	Java r14_x	
FIQ	`SUBS PC, R14_fiq, #4`	PC + 4	PC+4	PC+4	Where the PC is the address of the instruction that was not executed because the FIQ or IRQ took priority.
IRQ	`SUBS PC, R14_irq, #4`	PC + 4	PC+4	PC+4	
DABT	`SUBS PC, R14_abt, #8`	PC + 8	PC+8	PC+8	Where the PC is the address of the Load or Store instruction that generated the Data Abort.
RESET	`NA`	-	-	-	The value saved in r14_svc on reset is Unpredictable.
BKPT	`SUBS PC, R14_abt, #4`	PC + 4	PC+4	PC+4	Software breakpoint.

2.10.3 Entering an ARM exception

When handling an ARM exception the ARM11 MPCore processor:

1. Preserves the address of the next instruction in the appropriate LR. When the exception entry is from:

   ARM and Java states:

   The ARM11 MPCore processor writes the value of the PC into the LR, offset by a value (current PC + 4 or PC + 8 depending on the exception) that causes the program to resume from the correct place on return

   Thumb state:

   The ARM11 MPCore processor writes the value of the PC into the LR, offset by a value (current PC + 2, PC + 4 or PC + 8 depending on the exception) that causes the program to resume from the correct place on return.

   The exception handler does not have to determine the state when entering an exception. For example, in the case of a SWI, `MOVS PC, r14_svc` always returns to the next instruction regardless of whether the SWI was executed in ARM or Thumb state.

2. Copies the CPSR into the appropriate SPSR.

3. Forces the CPSR mode bits to a value that depends on the exception.

4. Forces the PC to fetch the next instruction from the relevant exception vector.

The ARM11 MPCore processor can also set the interrupt disable flags to prevent otherwise unmanageable nesting of exceptions.

——— **Note** ———

Exceptions are always entered, handled, and exited in ARM state. When the processor is in Thumb state or Java state and an exception occurs, the switch to ARM state takes place automatically when the exception vector address is loaded into the PC.

2.10.4 Leaving an ARM exception

When an exception has completed, the exception handler must move the LR, minus an offset to the PC. Table 2-5 on page 2-26 shows the type of exception and the offsets associated with it.

Typically the return instruction is an arithmetic or logical operation with the S bit set and rd = r15, so the core copies the SPSR back to the CPSR.

——— **Note** ———

The action of restoring the CPSR from the SPSR automatically resets the T bit and J bit to the values held immediately prior to the exception. The A, I, and F bits are also automatically restored to the value they held immediately prior to the exception.

2.10.5 Reset

When the **nCPURESET** signal is driven LOW a reset occurs, and the ARM11 MPCore processor abandons the executing instruction.

When **nRESETIN** is driven HIGH again the ARM11 MPCore processor:

1. Forces CPSR M[4:0] to b10011 (Supervisor mode), sets the A, I, and F bits in the CPSR, and clears the CPSR T bit and J bit. The E bit is set based on the state of the **BIGENDINIT** and **UBITINIT** pins. Other bits in the CPSR are indeterminate.
2. Forces the PC to fetch the next instruction from the reset vector address.
3. Reverts to ARM state, and resumes execution.

After reset, all register values except the PC and CPSR are indeterminate.

See Chapter 11 *Clocking, Resets, and Power Management* for more details of the reset behavior for the ARM11 MPCore processor.

2.10.6 Fast interrupt request

The *Fast Interrupt Request* (FIQ) exception supports fast interrupts. In ARM state, FIQ mode has eight private registers to reduce, or even remove the requirement for register saving (minimizing the overhead of context switching).

An FIQ is externally generated by taking the **nFIQ** signal input LOW. The **nFIQ** input is registered internally to the ARM11 MPCore processor. It is the output of this register that is used by the ARM11 MPCore processor control logic.

Irrespective of whether exception entry is from ARM state, Thumb state, or Java state, an FIQ handler returns from the interrupt by executing:

```
SUBS PC,R14_fiq,#4
```

You can disable FIQ exceptions within a privileged mode by setting the CPSR F flag. When the F flag is clear, the ARM11 MPCore processor checks for a LOW level on the output of the nFIQ register at the end of each instruction.

FIQs and IRQs are disabled when an FIQ occurs. You can use nested interrupts but it is up to you to save any corruptible registers and to re-enable FIQs and interrupts.

2.10.7 Interrupt request

The IRQ exception is a normal interrupt caused by a LOW level on the **nIRQ** input. IRQ has a lower priority than FIQ, and is masked on entry to an FIQ sequence.

Irrespective of whether exception entry is from ARM state, Thumb state, or Java state, an IRQ handler returns from the interrupt by executing:

```
SUBS PC,R14_irq,#4
```

You can disable IRQ exceptions within a privileged mode by setting the CPSR I flag. When the I flag is clear, the ARM11 MPCore processor checks for a LOW level on the output of the nIRQ register at the end of each instruction.

IRQs are disabled when an IRQ occurs. You can use nested interrupts but it is up to you to save any corruptible registers and to re-enable IRQs.

2.10.8 Aborts

An abort can be caused by either:

- the MMU signalling an internal abort
- an External Abort being raised from the AXI interfaces, by an AXI Slave or Decode Error response.

There are two types of abort:

- *Prefetch Abort*
- *Data Abort*.

IRQs are disabled when an abort occurs.

Prefetch Abort

This is signaled with the Instruction Data as it enters the pipeline Decode stage.

When a Prefetch Abort occurs, the ARM11 MPCore processor marks the prefetched instruction as invalid, but does not take the exception until the instruction is to be executed. If the instruction is not executed, for example because a branch occurs while it is in the pipeline, the abort does not take place.

After dealing with the cause of the abort, the handler executes the following instruction irrespective of the processor operating state:

```
SUBS PC,R14_abt,#4
```

This action restores both the PC and the CPSR, and retries the aborted instruction.

Data Abort

Data Abort on the ARM11 MPCore processor can be precise or imprecise. Precise Data Aborts are those generated after performing an instruction side CP15 operation, and all those generated by the MMU:

- alignment faults
- translation faults
- domain faults
- permission faults.

Data Aborts that occur because of watchpoints are imprecise in that the processor and system state presented to the abort handler is the processor and system state at the boundary of an instruction shortly after the instruction that caused the watchpoint (but before any following load/store instruction). Because the state that is presented is consistent with an instruction boundary, these aborts are restartable, even though they are imprecise.

Errors that cause externally generated Data Aborts, signaled by AXI **RRESP** or **BRESP**, might be precise or imprecise. Two separate FSR encodings indicate if the External Abort is precise or imprecise.

External Data Aborts are precise if:

- all aborts to loads or stores to Strongly Ordered memory are precise

- all aborts to loads to the Program Counter or the CPSR are precise
- all aborts on the load part of an SWP are precise
- all other External Aborts are imprecise.

External aborts are supported on cachable locations. The abort is transmitted to the processor only if a word requested by the processor had an External Abort.

Precise Data Aborts

A precise Data Abort is signaled when the abort exception enables the processor and system state presented to the abort handler to be consistent with the processor and system state when the aborting instruction was executed. With precise Data Aborts, the restarting of the processor after the cause of the abort has been rectified is straightforward.

The ARM11 MPCore processor implements the *base restored Data Abort model*, which differs from the *base updated Data Abort model* implemented by the ARM7TDMI-S.

With the *base restored Data Abort model*, when a Data Abort exception occurs during the execution of a memory access instruction, the base register is always restored by the processor hardware to the value it contained before the instruction was executed. This removes the requirement for the Data Abort handler to unwind any base register update, which might have been specified by the aborted instruction. This simplifies the software Data Abort handler. See the *ARM Architecture Reference Manual* for more details.

After dealing with the cause of the abort, the handler executes the following return instruction irrespective of the processor operating state at the point of entry:

```
SUBS PC,R14_abt,#8
```

This restores both the PC and the CPSR, and retries the aborted instruction.

Imprecise Data Aborts

An imprecise Data Abort is signaled when the processor and system state presented to the abort handler cannot be guaranteed to be consistent with the processor and system state when the aborting instruction was issued.

2.10.9 Imprecise Data Abort mask in the CPSR/SPSR

An imprecise Data Abort caused, for example, by an external error on a write that has been held in a Write Buffer, is asynchronous to the execution of the causing instruction and can occur many cycles after the instruction that caused the memory access has retired. For this reason, the imprecise Data Abort can occur at a time that the processor is in Abort mode because of a precise Data Abort, or can have live state in Abort mode, but be handling an interrupt.

To avoid the loss of the Abort mode state (r14 and SPSR_abt) in these cases, which leads to the processor entering an unrecoverable state, the existence of a pending imprecise Data Abort must be held by the system until a time when the Abort mode can safely be entered.

A mask is added into the CPSR to indicate that an imprecise Data Abort can be accepted. This bit is the A bit. The imprecise Data Abort causes a Data Abort to be taken when imprecise Data Aborts are not masked. When imprecise Data Aborts are masked, then the implementation is responsible for holding the presence of a pending imprecise Data Abort until the mask is cleared and the abort is taken.

The A bit is set automatically on entry into Abort Mode, IRQ, and FIQ Modes, and on Reset.

2.10.10 Software interrupt instruction

You can use the *software interrupt* (SWI) instruction to enter Supervisor mode, usually to request a particular supervisor function. The SWI handler reads the opcode to extract the SWI function number. A SWI handler returns by executing the following instruction, irrespective of the processor operating state:

```
MOVS PC, R14_svc
```

This action restores the PC and CPSR, and returns to the instruction following the SWI.

IRQs are disabled when a software interrupt occurs.

2.10.11 Undefined instruction

When an instruction is encountered that neither the ARM11 MPCore processor, nor any coprocessor in the system, can handle the ARM11 MPCore processor takes the Undefined instruction trap. Software can use this mechanism to extend the ARM instruction set by emulating Undefined coprocessor instructions.

After emulating the failed instruction, the trap handler executes the following instruction, irrespective of the processor operating state:

```
MOVS PC,R14_und
```

This action restores the CPSR and returns to the next instruction after the Undefined instruction.

IRQs are disabled when an Undefined instruction trap occurs. For more details about Undefined instructions, see the *ARM Architecture Reference Manual*.

2.10.12 Breakpoint instruction (BKPT)

A breakpoint (BKPT) instruction operates as though the instruction causes a Prefetch Abort.

A breakpoint instruction does not cause the ARM11 MPCore processor to take the Prefetch Abort exception until the instruction reaches the Execute stage of the pipeline. If the instruction is not executed, for example because a branch occurs while it is in the pipeline, the breakpoint does not take place.

After dealing with the breakpoint, the handler executes the following instruction irrespective of the processor operating state:

```
SUBS PC,R14_abt,#4
```

This action restores both the PC and the CPSR, and retries the breakpointed instruction.

Note

If the EmbeddedICE-RT logic is configured into halt mode, a breakpoint instruction causes the ARM11 MPCore processor to enter debug state. See *Halt mode debugging* on page 12-4.

2.10.13 Exception vectors

Table 2-6 shows the CP15 c1 Control Register V bit settings for configuring the location of the exception vector addresses.

Table 2-6 Configuration of exception vector address locations

Value of V bit	Exception vector base location
0	`0x00000000`
1	`0xFFFF0000`

Table 2-7 shows the exception vector addresses and entry conditions for the different exception types.

Table 2-7 Exception vectors

Exception	Offset from vector base	Mode on entry	A bit on entry	F bit on entry	I bit on entry
Reset	`0x00`	Supervisor	Disabled	Disabled	Disabled
Undefined instruction	`0x04`	Undefined	Unchanged	Unchanged	Disabled
Software interrupt	`0x08`	Supervisor	Unchanged	Unchanged	Disabled
Prefetch Abort	`0x0C`	Abort	Disabled	Unchanged	Disabled
Data Abort	`0x10`	Abort	Disabled	Unchanged	Disabled
Reserved	`0x14`	Reserved	-	-	-
IRQ	`0x18`	IRQ	Disabled	Unchanged	Disabled
FIQ	`0x1C`	FIQ	Disabled	Disabled	Disabled

2.10.14 Exception priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order that they are handled:

1. Reset, highest priority.
2. Precise Data Abort.
3. FIQ.
4. IRQ.
5. Imprecise Data Aborts.
6. Prefetch Abort.
7. BKPT, Undefined instruction, and SWI, lowest priority.

Some exceptions cannot occur together:

- The BKPT, or Undefined instruction, and SWI exceptions are mutually exclusive. Each corresponds to a particular, non-overlapping, decoding of the current instruction.
- When FIQs are enabled, and a precise Data Abort occurs at the same time as an FIQ, the ARM11 MPCore processor enters the Data Abort handler, and proceeds immediately to the FIQ vector.

  A normal return from the FIQ causes the Data Abort handler to resume execution.

Precise Data Aborts must have higher priority than FIQs to ensure that the transfer error does not escape detection. You must add the time for this exception entry to the worst-case FIQ latency calculations in a system that uses aborts to support virtual memory.

The FIQ handler must not access any memory that can generate a Data Abort, because the initial Data Abort exception condition is lost if this happens.

2.11 Additional instructions

To support extensions to ARMv6, the MPCore processor includes these additional instructions:

- Load Register Exclusive instructions, see *LDREXB*, *LDREXH* on page 2-37, and *LDREXD* on page 2-38
- Store Register Exclusive instructions, see *STREXB* on page 2-39, *STREXH* on page 2-41, and *STREXD* on page 2-42
- Clear Register Exclusive instruction, see *CLREX* on page 2-43
- True no operation instruction, see *True No Operation and Yield* on page 2-48
- *Pseudo-code functions for exclusive operations* on page 2-44
- *Send Event instruction* on page 2-45
- *Wait for Event WFE* on page 2-46
- *Wait for Interrupt WFI* on page 2-47
- *Pseudo-code functions for WFE and WFI* on page 2-47
- *Page table descriptors definition* on page 2-49
- *Access permission and ForceAP bit* on page 2-49.

2.11.1 LDREXB

Figure 2-8 shows the LDREXB format.

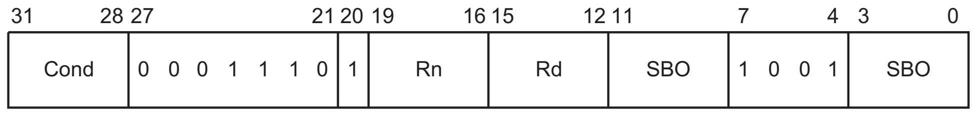


Figure 2-8 LDREXB instruction

LDREXB (Load Register Byte Exclusive) loads a byte from memory, zero-extends the byte to a 32-bit word, and either:

- If the physical address has the NonCachable TLB attribute, marks the address as exclusive access for the executing processor.
- Otherwise, causes the executing processor to tag the fact that it has an outstanding tagged physical access.

Syntax

Syntax is:

```
LDREXB{<cond>} <Rd>, [<Rn>];
```

Where:

`<cond>`	Is the condition under which the instruction is executed. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the memory byte addressed by `<Rd>`.
`<Rn>`	Specifies the register containing the address.

Operation

The LDREXB operation is as follows:

```
If ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    Rd = Memory[Rn,1]
    physical_address = TLB(Rn)
    If NonCachable(Rn) == 1 then
     MarkExclusiveGlobal(physical_address, processor_id, 1)
    MarkExclusiveLocal(physical_address,processor_id, 1)
```

No alignment restrictions apply to the addresses of this instruction.

2.11.2 LDREXH

Figure 2-9 shows the format of the Load Register Halfword Exclusive, LDREXH instruction.

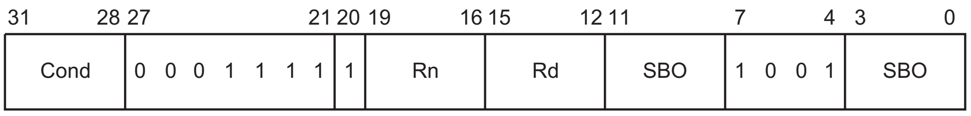


Figure 2-9 LDREXH instruction

LDREXH (Load Register Halfword Exclusive) loads a halfword from memory, zero-extends the byte to a 32-bit word, and either:

- If the physical address has the NonCachable TLB attribute, marks the address as exclusive access for the executing processor.
- Otherwise, causes the executing processor to tag the fact that it has an outstanding tagged physical access.

Syntax

Syntax is:

```
LDREXH{<cond>} <Rd>, [<Rn>];
```

Where:

`<cond>`	Is the condition under which the instruction is executed. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the memory halfword addressed by `<Rd>`.
`<Rn>`	Specifies the register containing the address.

Operation

The LDREXH operation is as follows:

```
If ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    Rd = Memory[Rn,2]
    physical_address = TLB(Rn)
    If NonCachable(Rn) == 1 then
     MarkExclusiveGlobal(physical_address, processor_id, 2)
    MarkExclusiveLocal(physical_address, processor_id, 2)
```

This instruction has a halfword alignment requirement, and generates alignment faults if it is not met if (A, U) == (0, 1), (1, 0) or (1, 1) in CP15 register 1.

2.11.3 LDREXD

Figure 2-10 shows the format of the Load Register Doubleword Exclusive, LDREXD, instruction.

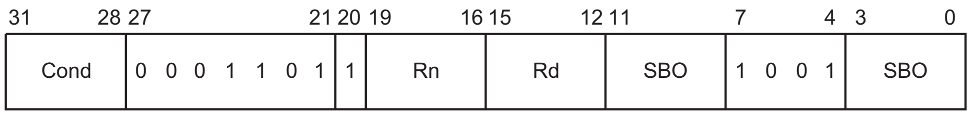


Figure 2-10 LDREXD instruction

LDREXD (Load Registers Doubleword Exclusive) loads a byte from memory, and either:

- If the physical address has the NonCachable TLB attribute, marks the address as exclusive access for the executing processor.
- Otherwise, causes the executing processor to tag the fact that it has an outstanding tagged physical access.

The pair of registers is restricted to being an even-numbered register and the odd-numbered register that immediately follows it, for example, R10 and R11.

Syntax

Syntax is:

```
LDREXD{<cond>} <Rd>, [<Rn>];
```

Where:

`<cond>`	Is the condition under which the instruction is executed. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the memory byte addressed by `<Rd>`.
`<Rn>`	Specifies the register containing the address.

Operation

The LDREXD operation is as follows:

```
If ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    Rd = Memory[Rn,8]
    physical_address = TLB(Rn)
    If NonCachable(Rn) == 1 then
       MarkExclusiveGlobal(physical_address, processor_id, 8)
    MarkExclusiveLocal(physical_address, processor_id, 8)
```

This instruction has a doubleword alignment requirement, and generates alignment faults if it is not met if (A, U) == (0, 1), (1, 0) or (1, 1) in CP15 register 1.

2.11.4 STREXB

Figure 2-11 shows the format of the Store Register Byte Exclusive, STREXB, instruction.

31 28	27 21	20	19 16	15 12	11	7 4	3 0
Cond	0 0 0 1 1 1 0	0	Rn	Rd	SBO	1 0 0 1	Rm

Figure 2-11 STREXB instruction

STREXB (Store Register Byte exclusive) conditionally stores a byte from the least significant byte of a register to memory. The store only occurs if the executing processor has exclusive access to the memory addressed.

Syntax

Syntax is:

```
STREXB{<cond>} <Rd>, <Rm>, [<Rn>];
```

Where:

`<cond>`	Is the condition under which the instruction is executed. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the returned status value. The value returned is: • 0 if the operation updates memory • 1 if the operation fails to update memory.
`<Rn>`	Specifies the register containing the address.

Operation

The STREXB operation is as follows:

```
If ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    physical_address = TLB(Rn)
    If IsExclusiveLocal(physical_address, processor_id, 1) then
        If NonCachable(Rn) == 1 then
            physical_address = TLB(Rn)
            If IsMarkExclusiveGlobal(physical_address, processor_id,1) then
                Memory[Rn,1] = Rm
                Rd = 0
                ClearByAddress(physical_address,1)
            else
                Rd = 1
        else
            Memory[Rn,1] = Rm
            Rd = 0
    else
        Rd = 1
    ClearExclusiveLocal(processor_id)
```

This instruction has no address alignment restrictions.

2.11.5 STREXH

Figure 2-12 shows the format of the Store Exclusive Register Halfword instruction

31 28	27 21	20	19 16	15 12	11	7 4	3 0
Cond	0 0 0 1 1 1 1	0	Rn	Rd	SBO	1 0 0 1	Rm

Figure 2-12 STREXH instruction

STREXH (Store exclusive register halfword) conditionally stores a halfword from the least significant halfword of a register to memory. The store only occurs if the executing processor has exclusive access to the memory addressed.

Syntax

Syntax is:

```
STREXH{<cond>} <Rd>, <Rm>, [<Rn>];
```

Where:

`<cond>`	Is the condition under which the instruction is executed. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the returned status value. The value returned is: • 0 if the operation updates memory • 1 if the operation fails to update memory.
`<Rm>`	Specifies the register containing the halfword to be stored to memory.
`<Rn>`	Specifies the register containing the address.

Operation

The STREXH operation is as follows:

```
If ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    physical_address = TLB(Rn)
    If IsExclusiveLocal(physical_address, processor_id, 2) then
        If NonCachable (Rn) == 1 then
            physical_address = TLB(Rn)
            If IsMarkExclusiveGlobal(physical_address, processor_id,2) then
                Memory[Rn,2] = Rm
            Rd = 0
            ClearByAddress(physical_address,2)
        else
            Rd = 1
    else
        Memory[Rn,2] = Rm
        Rd = 0
else
    Rd = 1
ClearExclusiveLocal(processor_id)
```

This instruction has a halfword alignment requirement, and generates alignment faults if it is not met if (A, U) == (0, 1), (1, 0) or (1, 1) in CP15 register 1.

2.11.6 STREXD

Figure 2-13 shows the format of the STREXD instruction.

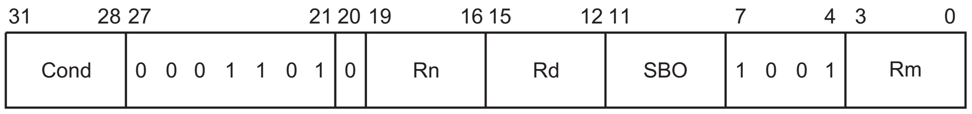


Figure 2-13 STREXD instruction

STREXD (Store register doubleword exclusive) conditionally stores a doubleword from a pair of ARM registers to two consecutive words of memory. The store only occurs if the executing processor has exclusive access to the memory addressed.

The pair of registers is restricted to being an even-numbered register and the odd-numbered register that immediately follows it (for example, R10 and R11).

Syntax

Syntax is:

`STREXD{<cond>} <Rd>, <Rm>, [<Rn>];`

Where:

`<cond>`	Is the condition under which the instruction is executed. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the returned status value. The value returned is: • 0 if the operation updates memory • 1 if the operation fails to update memory.

`<Rm>`	Specifies the register containing the doubleword to be stored to memory.
`<Rn>`	Specifies the register containing the address.

Syntax

The STREXD operation is as follows:

```
If ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    physical_address = TLB(Rn)
    If IsExclusiveLocal(physical_address, processor_id, 8) then
        If NonCachable(Rn) == 1 then
            If IsMarkExclusiveGlobal(physical_address, processor_id,8) then
                Memory[Rn,8] = Rm
                Rd = 0
                ClearByAddress(physical_address,8)
            else
                Rd = 1
        else
            Memory[Rn,8] = Rm
            Rd = 0
    else
        Rd = 1
    ClearExclusiveLocal(processor_id)
```

This instruction has a doubleword alignment requirement, and generates alignment faults if it is not met if (A, U) == (0, 1), (1, 0) or (1, 1) in CP15 register 1.

2.11.7 CLREX

CLREX (clear exclusive monitor) clears the local exclusive monitor

31 28	27 21	20	19 16	15 12	11	7 4	3 0
1 1 1 1	0 1 0 1 0 1 1	1	SBO	SBO	SBZ	0 0 0 1	SBO

Figure 2-14 CLREX instruction

Syntax

Syntax is:

```
CLREX;
```

Operation

CLREX operation is described below:

```
ClearExclusiveLocal(processor_id)
```

This operation is unconditional in the ARM instruction set.

2.11.8 Pseudo-code functions for exclusive operations

Pseudo-code functions for the exclusive access instructions are described below:

- TLB(Rn): If CP15 register 1 bit [0] (M bit) is set, TLB(Rn) returns the physical address corresponding to the virtual address in Rn for the executing processor's current process ID and TLB entries. If M Bit is not set, it returns the value in Rn.
- NonCachable(Rn): If CP15 register 1 bit [0] (M bit), NonCachable(Rn) will return the inverted value of the cachable memory region attribute (Shared non-coherent and non-shared non-cachable) corresponding to the virtual address in Rn for the executing processor's current process ID and TLB entries. If M bit is not set, it returns 0.
- ExecutingProcessor(): returns a value distinct amongst all processors in a given system, corresponding to the processor executing the operation.
- MarkExclusiveGlobal(physical_address, processor_id, size): records the fact that processor <processor_id> has requested exclusive access to a block of memory of an implementation defined size which at least covers size <size> bytes from address <physical_address>. It is Unpredictable whether this causes any previous request for exclusive access to any other address by the same processor to be cleared.
- MarkExclusiveLocal(physical_address, processor_id, size): records the fact that processor <processor_id> has requested exclusive access to a block of memory of an implementation defined size which at least covers size <size> bytes from address <physical_address>.
- IsExclusiveGlobal(physical_address, processor_id, size): returns TRUE if the processor <processor_id> has marked an implementation defined block of memory covering at least size <size> bytes from the address <physical_address> as exclusive access requested. Otherwise, it returns FALSE.
- IsExclusiveLocal(physical_address, processor_id, size): it returns TRUE if the processor <processor_id> has marked an implementation defined block of memory covering at least size <size> bytes from the address <physical_address> as exclusive access requested. Otherwise, it returns FALSE.

- ClearByAddress(physical_address, size): clears any request by any processor to mark an address including any of the bytes between <physical_address> and (<physical_address> + <size> -1) as an exclusive access.
- ClearExclusiveLocal(processor_id): clears the local record of processor <processor_id> that an address has had a request for an exclusive access.
- ClearExclusiveGlobal(processor_id): clears the global record of processor <processor_id> that an address has had request for an exclusive access.

2.11.9 Send Event instruction

SEV (Send Event) causes an event to be signaled to all CPUs within a multi-processor system. Figure 2-15 shows the SEV instruction format

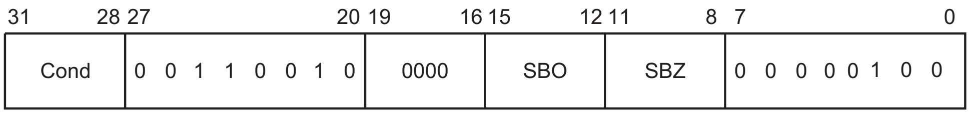


Figure 2-15 SEV instruction format

Figure 2-16 shows the SEV instruction format in Thumb state.

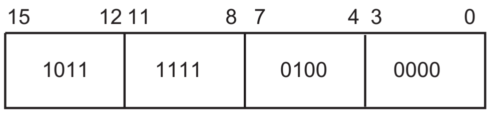


Figure 2-16 SEV instruction format in Thumb

Syntax is:

```
SEV(<cond>)
```

where:

`<cond>` Is the condition under which the instruction is to be executed.

SEV operation is described below:

```
if ConditionPassed(cond) then

SendEvent()
```

2.11.10 Wait for Event WFE

Figure 2-17 shows the Wait for Event instruction format in ARM state.

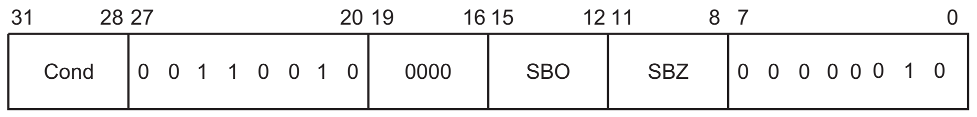


Figure 2-17 WFE instruction format

Figure 2-18 shows the Wait for Event instruction format in Thumb state.

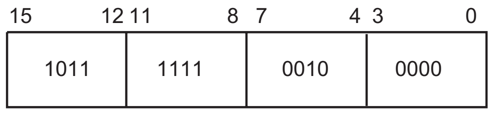


Figure 2-18 WFE format in Thumb mode

If the event register is currently set, WFE (Wait For Event) clears it and returns immediately. If the event register is not set, the processor suspends execution (Clock is stopped) until one of the following events take place:

- An IRQ interrupt, unless masked by the CPSR I Bit
- An FIQ interrupt, unless masked by the CPSR F Bit
- A Debug Entry request made to the processor and Debug is enabled
- An event is signaled by another processor using Send Event.
- An MP11 CPU return from exception.

Syntax is:

```
WFE(<cond>)
```

where:

`<cond>` Is the condition under which the instruction is to be executed.

WFE operation is described below:

```
if ConditionPassed(cond) then
  if EventRegistered() then
    ClearEventRegister()
  else
    WaitForEvent()
```

2.11.11 Wait for Interrupt WFI

Figure 2-19 shows the format of the Wait for Interrupt instruction in ARM state.

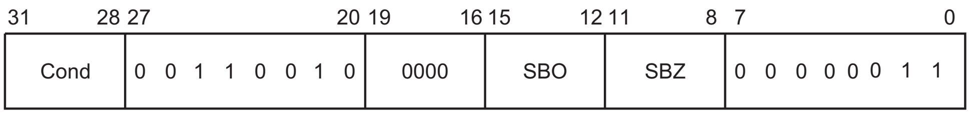


Figure 2-19 WFI instruction in ARM state

Figure 2-20 shows the format of the Wait for Interrupt instruction in Thumb state.

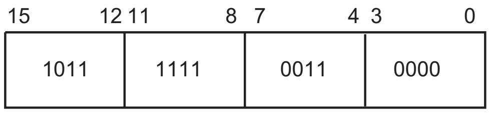


Figure 2-20 WFI instruction in Thumb state

WFI (Wait For Interrupt) makes the processor suspend execution (Clock is stopped) until one of the following events take place:

- An IRQ interrupt
- An FIQ interrupt
- A Debug Entry request made to the processor.

Syntax is:

```
WFI(<cond>)
```

where:

`<cond>`	Is the condition under which the instruction is to be executed.

WFI operation is described below:

```
if ConditionPassed(cond) then
    WaitForInterrupt()
```

2.11.12 Pseudo-code functions for WFE and WFI

Pseudo-code functions for the two Wait For Event related instructions and the redefined Wait For Interrupt instruction are:

- SendEvent() causes an event on the processor to be asserted. It has no other effect.

- EventRegistered() returns TRUE if the event register on the current processor is set. The event register is set on a WakeUpAction, a SendEvent() from any MP11 CPU, or an external event coming from **EVNTIN**.
- ClearEventRegister() causes the event register on the current processor to be reset.
- WaitForEvent() causes the processor to suspend execution until a WakeUpAction is observed by that processor, or an Event is sent by any processor as a result of that processor executing a SendEvent().
- WaitForInterrupt() causes the processor to suspend execution until an UnMaskedWakeUpAction is observed by that processor.
- A WakeUpAction is one of the following events:
  - — An IRQ interrupt, unless masked by the CPSR I Bit
  - — An FIQ interrupt, unless masked by the CPSR F Bit
  - — A Debug Entry request made to the processor and Debug is enabled.
  - — An MP11 CPU return from exception.
- An UnMaskedWakeUpAction is one of the following events:
  - — An IRQ interrupt
  - — An FIQ interrupt
  - — A Debug Entry request made to the processor.

2.11.13 True No Operation and Yield

Figure 2-21 shows the format of the No Operation, NOP32, instruction

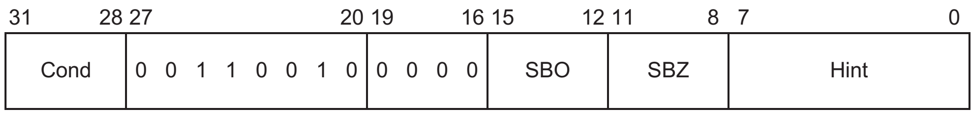


Figure 2-21 NOP32

Syntax is:

```
NOP32(<cond>) <hint>
```

where:

`<cond>` Is the condition under which the instruction executes. It produces no useful change in functionality, but is provided to ensure disassembly followed by reassembly always regenerates the original code.

`<Hint>`	Default to zero
	Hint == 0 true NOP
	Hint == 1 YIELD

All others values except the ones defined for SEV, WFE and WFI instructions are RESERVED and execute as a NOP.

The instruction acts as a NOP irrespective of whether the condition passes or fails, effectively the ALWAYS condition.

2.11.14 Page table descriptors definition

To free up bits TEX[2:1] of the page table descriptors for software usage, remap capability has been added. This behavior is only enabled when CP15 register 1, bit [28] is set, so providing backward compatibility with ARMv6. See *Page table descriptors when using remapping* on page 5-18 and *c10, Memory remap registers* on page 3-60.

2.11.15 Access permission and ForceAP bit

To optimize memory management algorithms in the OS an Access Bit has been introduced. It is only enabled when CP15 register 1, bit [29] is set. See *Access permission and ForceAP bit* on page 5-13.

Chapter 3
Control Coprocessor CP15

This chapter describes the MPCore control coprocessor CP15 registers and how they are accessed. It also provides information for programming the microprocessor. It contains the following sections:

3.1 About control coprocessor CP15

The control coprocessor, CP15, implements a range of control functions and provides status information for the ARM11 MPCore processor. The main functions controlled by CP15 are:

- overall system control and configuration of the ARM11 MPCore processor
- cache configuration and management
- *Memory Management Unit* (MMU) configuration and management
- system performance monitoring.

3.1.1 Accessing CP15 registers

You can access CP15 registers with MRC and MCR instructions. Figure 3-1 shows the instruction bit pattern of MRC and MCR instructions.

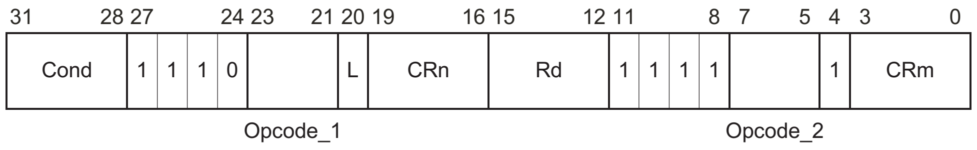


Figure 3-1 CP15 MRC and MCR bit pattern

The assembler for these instructions is:

```
MRC{cond} P15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>
MCR{cond} P15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>
```

Instructions CDP, LDC, and STC, together with unprivileged MRC and MCR instructions to privileged-only CP15 locations, cause the Undefined instruction trap to be taken. The CRn field of MRC and MCR instructions specifies the coprocessor register to access. The CRm field and Opcode_2 fields specify a particular action when addressing registers. The L bit distinguishes between an MRC (L=1) and an MCR (L=0).

Note

Attempting to read from a nonreadable register, or to write to a nonwritable register causes Unpredictable results.

The Opcode_1, Opcode_2, and CRm fields Should Be Zero in all instructions that access CP15, except when the values specified are used to select desired operations. Using other values results in Unpredictable behavior.

In all cases, reading from or writing any data values to any CP15 registers, including those fields specified as *Unpredictable* (UNP), *Should Be One* (SBO), or *Should Be Zero* (SBZ), does not cause any physical damage to the chip.

3.2 CP15 registers arranged by function

Table 3-1 shows the system functions of the CP15 control registers.

Table 3-1 CP15 register functions

Function	Register	Reference to description
Overall system configuration and control	Control	See *c1, Control Register* on page 3-30
	Auxiliary Control	See *c1, Auxiliary Control Register* on page 3-34
	Coprocessor Access Control	See *c1, Coprocessor Access Control Register* on page 3-36
	ID Code	See *c0, ID Code Register* on page 3-12
	CPU ID	See *c0, CPU ID Register* on page 3-15
	Thread ID	See *c13, Thread ID registers* on page 3-68
	Processor Feature 0	See *Processor Feature Register 0, ID_PFR0* on page 3-17
	Processor Feature 1	See *Processor Feature Register 1, ID_PFR1* on page 3-17
	Debug Feature 0	See *Debug Feature Register 0, ID_DFR0* on page 3-18
	Memory Model 0	See *Memory Model Features Register 0, ID_MMFR0* on page 3-19
	Memory Model 1	See *Memory Model feature register 1 (ID_MMFR1)* on page 3-20
	Memory Model 2	See *Memory Model feature register 2 (ID_MMFR2)* on page 3-21
	Memory Model 3	See *Memory Model Feature Register 3 (ID_MMFR2)* on page 3-23
	Instruction Set Attributes 0	See *Instruction Set Attributes Register 0 (ID_ISAR0)* on page 3-25
	Instruction Set Attributes 1	See *Instruction Set Attributes Register 1 (ID_ISAR1)* on page 3-26
	Instruction Set Attributes 2	See *Instruction Set Attributes Register 2 (ID_ISAR2)* on page 3-27
	Instruction Set Attributes 3	See *Instruction Set Attributes Register 3 (ID_ISAR3)* on page 3-28
	Instruction Set Attributes 4	See *Instruction Set Attributes Register 4 (ID_ISAR4)* on page 3-29

Table 3-1 CP15 register functions (continued)

Function	Register	Reference to description
Cache and TLB operation and control	Cache Type	See *c0, Cache Type Register* on page 3-12
	Cache operations	See *c7, Cache Operations Register* on page 3-46
	Data Cache Lockdown	See *c9, Data Cache Lockdown Register* on page 3-58
	Domain Access Control	See *c3, Domain Access Control Register* on page 3-41
	TLB Type	See *c0, TLB Type Register* on page 3-14
	TLB Operations	See *c8, TLB Operations Register* on page 3-55
	TLB Lockdown	See *c10, TLB Lockdown Register* on page 3-60
	Branch Target Cache Operations	See *c7, Cache Operations Register* on page 3-46
Access to main TLB lockdown entries	TLB Lockdown Attribute	See *c15, TLB lockdown operations* on page 3-75
	TLB Lockdown Entry	See *c15, TLB lockdown operations* on page 3-75
	TLB Lockdown PA	See *c15, TLB lockdown operations* on page 3-75
	TLB Lockdown VA	See *c15, TLB lockdown operations* on page 3-75
TLB Debug	TLB Debug Control	See *c15, TLB Debug Control Register* on page 3-73

Table 3-1 CP15 register functions (continued)

Function	Register	Reference to description
Memory Management Unit configuration and control	Context ID	See *c13, Context ID Register* on page 3-67
	FCSE PID	See *c13, FCSE PID Register* on page 3-64
	Data Fault Address	See *c6, Fault Address Register* on page 3-45
	Data Fault Status	See *c5, Data Fault Status Register* on page 3-42
	Watchpoint Fault Address	See *c6, Watchpoint Fault Address Register* on page 3-45
	Instruction Fault Status	See *c5, Instruction Fault Status Register* on page 3-44
	Domain Access Control	See *c3, Domain Access Control Register* on page 3-41
	Translation Table Base Control	See *c2, Translation Table Base Control Register* on page 3-40
	Translation Table Base 0	See *c2, Translation Table Base Register 0* on page 3-37
	Translation Table Base 1	See *c2, Translation Table Base Register 1* on page 3-39
	Memory Remap	See *c10, Memory remap registers* on page 3-60
System performance monitoring	Performance Monitor Control	See *c15, Performance Monitor Control Register (PMNC)* on page 3-68
	Count 0 (PMN0)	See *c15, Count Register 0 (PMN0)* on page 3-73
	Count 1 (PMN1)	See *c15, Count Register 1 (PMN1)* on page 3-73
	Cycle Counter (CCNT)	See *c15, Cycle Counter Register (CCNT)* on page 3-72

3.3 Summary of control coprocessor CP15 registers and operations

Table 3-2 shows the registers and operations described in this section, arranged numerically.

Table 3-2 Summary of CP15 registers and operations

CRn	Op1[a]	CRm	Op2[a]	Register/operation name	Type[b]	Reset value	Page
c0	0	c0	0	ID Code	RO	`0x410FB022`	page 3-12
			1	Cache Type	RO	Implementation -defined[c]	page 3-12
			3	TLB Type	RO	`0x00000800`	page 3-14
			5	CPU ID	RO	Implementation -defined[c]	page 3-15
		c1	0	Processor Feature Register 0	RO	`0x00000111`	page 3-16
			1	Processor Feature Register 1	RO	`0x00000001`	page 3-16
			2	Debug Feature Register 0	RO	`0x00000002`	page 3-16
			4	Memory Model Feature Register 0	RO	`0x01100103`	page 3-16
			5	Memory Model Feature Register 1	RO	`0x10020302`	page 3-16
			6	Memory Model Feature Register 2	RO	`0x01222000`	page 3-16
			7	Memory Model Feature Register 3	RO	`0x00000000`	page 3-16
		c2	0	ISA Feature Register 0	RO	`0x00100011`	page 3-24
			1	ISA Feature Register 1	RO	`0x12002111`	page 3-24
			2	ISA Feature Register 2	RO	`0x11221011`	page 3-24
			3	ISA Feature Register 3	RO	`0x01102131`	page 3-24
			4	ISA Feature Register 4	RO	`0x00000141`	page 3-24
c1	0	c0	0	Control	R/W	`0x00054078`	page 3-30
			1	Auxiliary Control	R/W	`0x0000000F`	page 3-34
			2	Coprocessor Access Control	R/W	`0x00000000`	page 3-36

Table 3-2 Summary of CP15 registers and operations (continued)

CRn	Op1[a]	CRm	Op2[a]	Register/operation name	Type[b]	Reset value	Page
c2	0	c0	0	Translation Table Base 0	R/W	`0x00000000`	page 3-37
			1	Translation Table Base 1	R/W	`0x00000000`	page 3-39
			2	Translation Table Base Control	R/W	`0x00000000`	page 3-40
c3	0	c0	0	Domain Access Control	R/W	`0x00000000`	page 3-41
c4	-	-	-	Not used	-	-	-
c5	0	c0	0	Data Fault Status	R/W	-	page 3-42
			1	Instruction Fault Status	R/W	-	page 3-44
c6	0	c0	0	Data Fault Address	R/W	-	page 3-45
			1	Watchpoint Fault Address	R/W	-	page 3-45

Table 3-2 Summary of CP15 registers and operations (continued)

CRn	Op1[a]	CRm	Op2[a]	Register/operation name	Type[b]	Reset value	Page
c7	0	c0	4	WFI	WO	-	page 3-49
		c4	0	PA Register	RO	`0x00000000`	page 3-53
		c5	0	Invalidate Entire Instruction Cache	WO	-	page 3-46
			1	Invalidate Instruction Cache (using MVA)	WO	-	page 3-46
			2	Invalidate Instruction Cache (using Index)	WO	-	page 3-46
			4	Flush Prefetch Buffer	**WO**	-	page 3-46
			6	Flush Entire Branch Target Cache	WO	-	page 3-46
			7	Flush Branch Target Cache Entry	WO	-	page 3-46
		c6	0	Invalidate Entire Data Cache	WO	-	page 3-46
			1	Invalidate Data Cache Line (using MVA)	WO	-	page 3-46
			2	Invalidate Data Cache Line (using Index)	WO	-	page 3-46
		c7	0	Invalidate Both Caches	WO	-	page 3-46
		c8	0	VA to PA privileged read	WO	-	page 3-52
			1	VA to PA privileged write	WO	-	page 3-52
			2	VA to PA user read	WO	-	page 3-52
			3	VA to PA user write	WO	-	page 3-52
		c10	0	Clean Entire Data Cache	WO	-	page 3-46
			1	Clean Data Cache Line (using MVA)	WO	-	page 3-46
			2	Clean Data Cache Line (using Index)	WO	-	page 3-46
			4	Data Synchronization Barrier	**WO**	-	page 3-46
			5	Data Memory Barrier	**WO**	-	page 3-46

Table 3-2 Summary of CP15 registers and operations (continued)

CRn	Op1[a]	CRm	Op2[a]	Register/operation name	Type[b]	Reset value	Page
c7	0	c14	0	Clean and Invalidate Entire Data Cache	WO	-	page 3-46
			1	Clean and Invalidate Data Cache Line (using MVA)	WO	-	page 3-46
			2	Clean and Invalidate Data Cache Line (using Index)	WO	-	page 3-46
c8	0	c5	0	Invalidate Instruction TLB	WO	-	page 3-55
			1	Invalidate Instruction TLB Single Entry	WO	-	page 3-55
			2	Invalidate Instruction TLB Entry on ASID match	WO	-	page 3-55
			3	Invalidate Instruction TLB Single Entry on MVA only	WO	-	page 3-58
		c6	0	Invalidate Data TLB	WO	-	page 3-55
			1	Invalidate Data TLB Single Entry	WO	-	page 3-55
			2	Invalidate Data TLB Entry on ASID match	WO	-	page 3-55
			3	Invalidate Data TLB Single Entry on MVA only	WO	-	page 3-58
		c7	0	Invalidate Unified TLB	WO	-	page 3-55
			1	Invalidate Unified TLB Single Entry	WO	-	page 3-55
			2	Invalidate Unified TLB Entry on ASID match	WO	-	page 3-55
			3	Invalidate Unified TLB Single Entry on MVA only	WO	-	page 3-58
c9	0	c0	0	Data Cache Lockdown	R/W	`0xFFFFFFF0`	page 3-58
c10	0	c0	0	TLB Lockdown	R/W	`0x00000000`	page 3-60
		c2	0	Primary Region Remap Register	R/W	`0x00098AA4`	page 3-60
			1	Normal Region Remap Register	R/W	`0x44E048E0`	page 3-60
c11	-	-	-	Not used	-	-	-

Table 3-2 Summary of CP15 registers and operations (continued)

CRn	Op1[a]	CRm	Op2[a]	Register/operation name	Type[b]	Reset value	Page
c12	-	-	-	Not used	-	-	-
c13	0	c0	0	FCSE PID	R/W	`0x00000000`	page 3-64
			1	Context ID	R/W	`0x00000000`	page 3-67
			2	Thread ID Register User and Privileged R/W accessible	R/W	`0x00000000`	page 3-68
			3	Thread ID Register User read-only and Privileged R/W accessible	R/W	`0x00000000`	page 3-68
			4	Thread ID Register Privileged R/W accessible only	R/W	`0x00000000`	page 3-68
c14	-	-	-	Not used	-	-	-
c15	0	c12	0	Performance Monitor Control	R/W	`0x00000000`	page 3-68
			1	Cycle Counter (CCNT)	R/W	Unpredictable	page 3-72
			2	Count 0 (PMN0)	R/W	Unpredictable	page 3-73
			3	Count 1 (PMN1)	R/W	Unpredictable	page 3-73
	5	c4	2	Read Main TLB Lockdown Entry	WO	-	page 3-75
			4	Write Main TLB Lockdown Entry	WO	-	page 3-75
		c5	2	Main TLB Lockdown VA	R/W	`0x00000000`	page 3-75
		c6	2	Main TLB Lockdown PA	R/W	`0x00000000`	page 3-75
		c7	2	Main TLB Lockdown Attribute	R/W	`0x00000000`	page 3-75
	7	c1	0	TLB Debug Control Register	R/W	`0x00000000`	page 3-74

a. Op1 = Opcode_1, Op2 = Opcode_2.
b. **Bold** text denotes that the register can be accessed in User mode.
c. The cache type reset value is determined by the size of the caches implemented.

3.4 Register descriptions

This section contains descriptions of all the CP15 registers arranged in numerical order, as shown in Table 3-2 on page 3-7.

3.4.1 c0, ID Code Register

The purpose of the ID Code Register is to return the device ID code that contains information about the processor. Processor ID information is defined across an ID Code Register and some Feature Version registers

The ID Code Register is a read-only register that can be accessed with the following CP15 instruction:

```
MRC p15,0,<Rd>,c0,c0,0; reads Main ID register
```

The contents of the ID Code register are shown in Figure 3-2.This is a read-only register that returns a 32-bit device ID code.

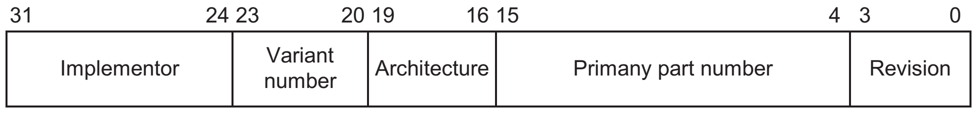


Figure 3-2 Register 0, ID Code

Table 3-3 shows the function of the bits in the Main ID Code Register.

Table 3-3 Register 0, ID Code

Register bits	Function	Value
[31:24]	Implementor	`0x41`
[23:20]	Variant number	`0x0`
[19:16]	Architectural Format description	`0xF`
[15:4]	Part number	`0xB02`
[3:0]	Revision number	`0x2`

3.4.2 c0, Cache Type Register

The purpose of the Cache Type Register is to provide information about the size and architecture of the cache for the operating system. This enables the operating system to establish how to clean the cache and how to lock it down.

The Cache Type Register is:

- in CP15 c0
- a 32-bit read only register
- accessible in privileged modes only.

You can access the Cache Type Register by reading CP15 c0 with the Opcode_2 field set to 1:

```
MRC p15,0,<Rd>,c0,c0,1; returns cache details
```

Figure 3-3 shows the format of the Cache Type Register.

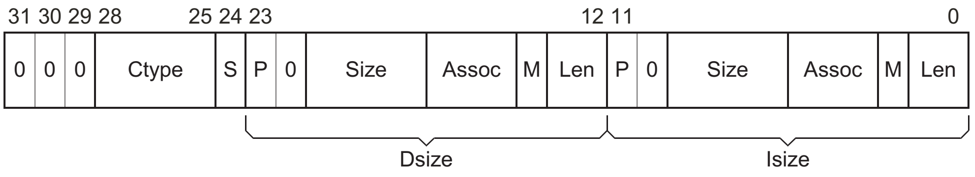


Figure 3-3 Cache Type Register format

Table 3-4 shows the Cache Type Register bit assignments.

Table 3-4 Cache Type Register bit assignment

Bit	Name	Function
[31:29]	-	SBZ/RAZ
[28:25]	Ctype	For MP11 CPUs, Ctype = b1110 MP11 CPUs support: • write back cache • Format C cache lockdown • Register 7 cache cleaning operations.

Table 3-4 Cache Type Register bit assignment (continued)

Bit	Name	Function
[24]	S Bit	Specifies if the cache is a unified cache (S = 0). For MP11 CPUs, S = 1.
[23:12]	DSize	Specifies the size, line length and associativity of the data cache. [23] = b0 (no page coloring) [22:21] = b00. [20:18] = size b101 = 16KB b110 = 32KB b111 = 64KB [17:15] = b010 (4-way associative) [14] = b0. [13:12] = b10 (8 words per cache line).
[11:0]	ISize	Specifies the size, line length and associativity of the data cache. [11] = b0 for 16KB (no page coloring) (Set to b1 for 32KB and 64KB instruction caches) [10:9] = b00. [8:6] = size b101 = 16KB b110 = 32KB b111 = 64KB [5:3] = b010 (4-way associative) [2] = b0. [1:0] = b10 (8 words per cache line).

3.4.3 c0, TLB Type Register

The purpose of the TLB Type Register is to return the number of lockable entries for the TLB

The TLB has 64 entries organized as a unified two-way set associative TLB. In addition, it has eight lockable entries, as specified by the read-only TLB Type Register.

The TLB Type Register is:

- in CP15 c0
- a 32-bit read only register

- accessible in privileged modes only.

You can access the TLB Type Register by reading CP15 c0 with the Opcode_2 field set to 3. For example:

```
MRC p15,0,<Rd>,c0,c0,3; returns TLB details
```

Figure 3-4 shows the format of the TLB Type Register.

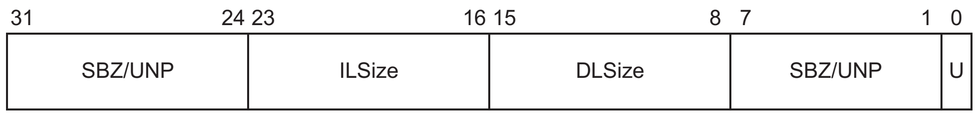


Figure 3-4 TLB Type Register format

Table 3-5 shows the functions of the bits in TLB Type Register.

Table 3-5 TLB Type Register field descriptions

Bits	Field	Description
[31:24]	SBZ/UNP	-
[23:16]	ILsize	Specifies the number of instruction TLB lockable entries. For ARM11 MPCore processors this is 0.
[15:8]	DLsize	Specifies the number of unified or data TLB lockable entries. For ARM11 MPCore processors this is 8.
[7:1]	SBZ/UNP	-
[0]	U	Specifies if the TLB is unified (0), or if there are separate instruction and data TLBs (1). For ARM11 MPCore processors this is 0.

3.4.4 c0, CPU ID Register

The purpose of the CPU ID Register is to

The CPU ID Register is:

- in CP15 c0
- a 32-bit read only register
- accessible in privileged modes only.

You can access the CPU ID Register by reading CP15 c0 with the following CP15 instruction:

```
MRC p15,0,<Rd>,c0,c0,5; returns CPU ID register
```

Figure 3-5 shows the format of the CPU ID Register.

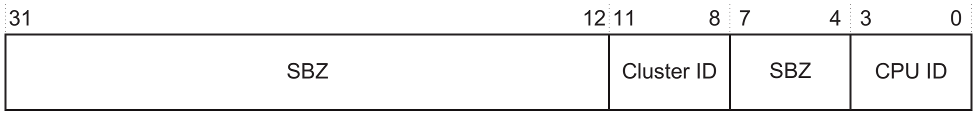


Figure 3-5 CPU ID Register format

The Cluster ID field value is set by the **CLUSTERID** configuration pins.

The CPU ID field value is different for each MP11 CPU in the ARM11 MPCore processor. Depending on the number of configured MP11 CPUs, the MP11 CPU IDs are:

- One MP11 CPU, the CPU ID is `0x0`.
- Two MP11 CPUs, the CPU IDs are `0x0 and 0x1`.
- Three MP11 CPUs, the CPU IDs are 0x0, 0x1, and `0x2`.
- Four MP11 CPUs, the CPU ID is `0x0, 0x1, 0x2, and 0x3`.

3.4.5 c0, Feature registers

This section describes feature registers:

- Processor feature registers
- Debug feature register
- Memory model registers.

```
MRC p15,0,<Rd>,c0,c1,{0-7}  ; reads feature version registers
```

Depending on the Opcode_2 value, the accessed register is:

- Opcode_2 = 0: ID_PFR0, Processor Feature Register 0
- Opcode_2 = 1: ID_PFR1, Processor Feature Register 1
- Opcode_2 = 2: ID_DFR0, Debug Feature Register 0
- Opcode_2 = 3: Reserved
- Opcode_2 = 4: IDMMFR0, Memory Model Feature Register 0
- Opcode_2 = 5: IDMMFR0, Memory Model Feature Register 1
- Opcode_2 = 6: IDMMFR0, Memory Model Feature Register 2
- Opcode_2 = 7: IDMMFR0, Memory Model Feature Register 3.

The RESERVED Opcode_2 value, Opcode_2=3, reads as zero.

Processor Feature Register 0, ID_PFR0

The purpose of the Processor Feature Register 0 is to provide information about the execution state support and programmer's model for the processor.

Processor Feature Register 0 is:

- in CP15 c0
- a 32-bit read-only register
- accessible in privileged modes only.

Figure 3-6 shows the format of the ID_PFR0 Register.

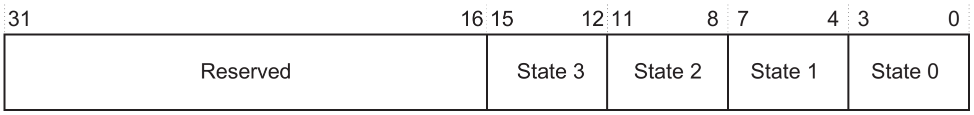


Figure 3-6 ID_PFR0 format

Table 3-6 describes the meaning of the ID_PFR0 bits.

Table 3-6 ID_PFRO bit assignments

Bit	Name	Value
[31:16]	RESERVED	RAZ
[15:12]	State 3 (J == 1, T == 1)	`0x0` (Not supported)
[11:8]	State 2 (J == 1, T == 0)	`0x1` (Java extension interface supported)
[7:4]	State 1 (J == 0, T == 1)	`0x1` (Thumb-1 encoding with all Thumb-1 basic instructions supported)
[3:0]	State 0 (J == 0, T == 0)	`0x1` (32-bit ARM instruction Set supported)

Processor Feature Register 1, ID_PFR1

The purpose of the Processor Feature Register 1 is to provide information about the execution state support and programmer's model for the processor.

Processor Feature Register 1 is:

- in CP15 c0
- a 32-bit read-only register common to the Secure and Non-Secure worlds
- accessible in privileged modes only.

Figure 3-7 on page 3-18 shows the format of the ID_PFR0 Register.

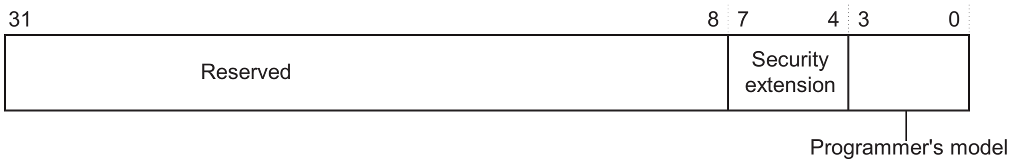


Figure 3-7 ID_PFR1 format

Table 3-7 describes the meaning of the ID_PFR1 bits.

Table 3-7 ID_PFR1 bit assignment

Bit	Name	Value
[31:8]	RESERVED	RAZ
[7:4]	Security extension	`0x0` (not supported)
[3:0]	Programmer's model	`0x1` (standard - ARMv4 - programmer's model

Debug Feature Register 0, ID_DFR0

The purpose of the Debug Feature Register 0 is to provide information about the debug system for the processor.

Debug Feature Register 0 is:
- in CP15 c0
- a 32-bit read-only register
- accessible in privileged modes only.

Figure 3-8 shows the format of the ID_DFR0 Register.

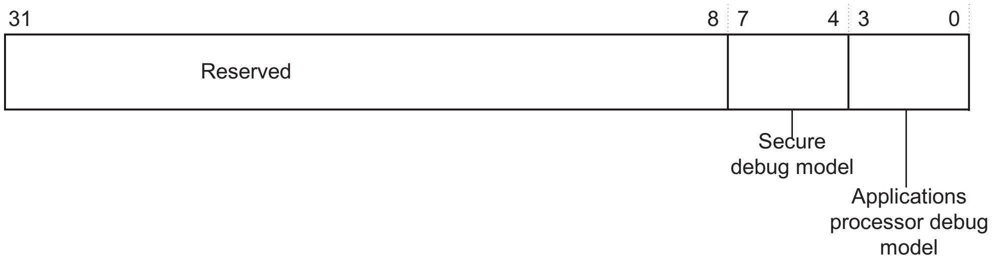


Figure 3-8 ID_DFR0 format

Table 3-8 describes the meaning of the ID_DFR0 bits.

Table 3-8 Debug Feature Register bit assignment

Bit	Name	Value
[31:8]	RESERVED	RAZ
[7:4]	Secure Debug Model	`0x0` (not supported)
[3:0]	Applications Processor Debug Model	`0x2` (v6 debug model - CP14 based)

Memory Model Features Register 0, ID_MMFR0

The purpose of the Memory Model Feature Register 0 is to provide information about the memory model, memory management, cache support, and TLB operations of the processor.

The Memory Model Feature Register 0 is:
- in CP15 c0
- a 32-bit read-only register common
- accessible in privileged modes only.

Figure 3-9 shows the bit arrangement for Memory Model Feature Register 0.

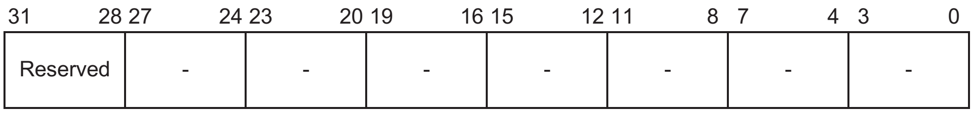


Figure 3-9 Memory Model Feature Register 0 format

Table 3-9 describes the meaning of the ID_MMFR0 bits.

Table 3-9 Memory Model Features Register 0 bit assignment

Bit	Name	Value
[31:28]	RESERVED	RAZ
[27:24]	FCSE	`0x1` (FCSE supported)
[23:20]	Auxiliary control register	`0x1` (ARMv6)

Table 3-9 Memory Model Features Register 0 bit assignment (continued)

Bit	Name	Value
[19:16]	TCM and associated DMA support	`0x0` (not supported)
[15:12]	Cache coherence support - DMA agent, shared memory	`0x0` (no shared supported)
[11:8]	Cache coherence support - CPU agent, shared memory	`0x1` (L1 cache shared support)
[7:4]	PMSA support	`0x0` (not supported)
[3:0]	VMSA support	`0x3` (VMSAv6 + Advanced OS Features)

Memory Model feature register 1 (ID_MMFR1)

The purpose of the Memory Model Feature Register 1 is to provide information about the memory model, memory management, cache support, and TLB operations of the processor.

The Memory Model Feature Register 1 is:

- in CP15 c0
- a 32-bit read-only register
- accessible in privileged modes only.

Figure 3-10 shows the bit arrangement for Memory Model Feature Register 1.

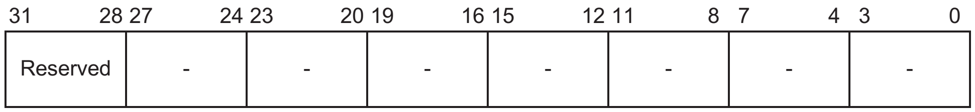


Figure 3-10 Memory Model Feature Register 1 format

Table 3-10 describes the meaning of the ID_MMFR1 bits.

Table 3-10 ID_MMFR1 bit assignments

Bits	Name	Value
[31:28]	Branch target buffer	`0x1` (requires flushing on VA change)
[27:24]	L1 test and clean operation on data cache (Harvard or unified)	`0x0` (not supported)
[23:20]	L1 cache (all) maintenance operation (unified)	`0x0` (not supported)
[19:16]	L1 cache (all) maintenance operation (Harvard)	`0x2` (Invalidation instruction cache, data cache and both)
[15:12]	L1 cache line maintenance operation by Set/Way (unified)	`0x0` (not supported)
[11:8]	L1 cache line maintenance operation by Set/Way (Harvard)	`0x3` (Clean data cache line, Clean and Invalidate data cache line, Invalidate data cache line, Invalidate instruction cache line)
[7:4]	L1 cache line maintenance operation by VA (unified)	`0x0` (not supported)
[3:0]	L1 cache line maintenance operation by VA (Harvard)	`0x2` (Clean data cache line, Invalidate data cache line, Invalidate instruction cache line, Clean and Invalidate data cache line, Invalidate BTAC line)

Memory Model feature register 2 (ID_MMFR2)

The purpose of the Memory Model Feature Register 2 is to provide information about the memory model, memory management, cache support, and TLB operations of the processor.

The Memory Model Feature Register 2 is:

- in CP15 c0
- a 32-bit read-only register common to the Secure and Non-Secure worlds
- accessible in privileged modes only.

Figure 3-11 on page 3-22 shows the bit arrangement for Memory Model Feature Register 2.

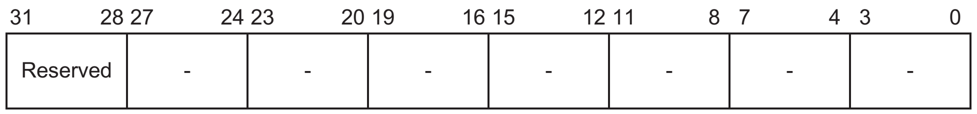


Figure 3-11 Memory Model Feature Register 2 format

Table 3-11 describes the meaning of the ID_MMFR2 bits.

Table 3-11 ID_MMFR2 bit assignments

Bit	Name	Value
[31:28]	Reserved	RAZ
[27:24]	Wait for interrupt stalling	`0x1` (wait for interrupt supported)
[23:20]	Memory barrier features	`0x2` (DataWriteBarrier, PrefetchFlush, DataMemoryBarrier)
[19:16]	TLB maintenance operation (unified)	`0x2` (Invalidate all entries, Invalidate TLB entry by MVA, Invalidate TLB entries by ASID match)
[15:12]	TLB maintenance operation (Harvard)	`0x2` (Invalidate all entries, Invalidate TLB entry by MVA, Invalidate TLB entries by ASID match) 0x2 (supported)
[11:8]	L1 cache maintenance range operation (Harvard)	`0x0` (not supported)
[7:4]	L1 background prefetch cache range operations (Harvard)	`0x0` (not supported)
[3:0]	L1 foreground prefetch cache range operation (Harvard)	`0x0` (not supported)

Memory Model Feature Register 3 (ID_MMFR2)

The purpose of the Memory Model Feature Register 3 is to provide information about the memory model, memory management, cache support, and TLB operations of the processor.

The Memory Model Feature Register 3 is:

- in CP15 c0
- a 32-bit read-only register
- accessible in privileged modes only.

Figure 3-12 shows the bit arrangement for Memory Model Feature Register 3.

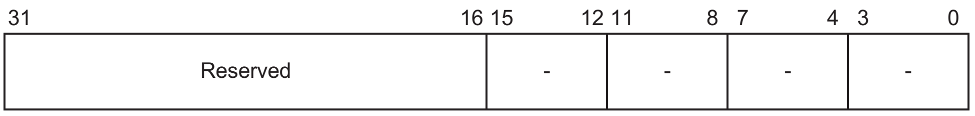


Figure 3-12 Memory Model Feature Register 3 format

Table 3-12 describes the meaning of the ID_MMFR3 bits.

Table 3-12 ID_MMFR3 bit assignments

Bit	Name	Value
[31:16]	Reserved	RAZ
[15:12]	L2 cache (all) maintenance operation (unified)	`0x0` (not supported)

Table 3-12 ID_MMFR3 bit assignments

Bit	Name	Value
[11:8]	L2 cache line maintenance operation with Set/Way (unified)	`0x0` (not supported)
[7:4]	L2 cache line maintenance operation with VA (unified)	`0x0` (not supported)
[3:0]	L2 cache line maintenance operation with PA (unified)	`0x0` (not supported)

3.4.6 c0, Instruction set attributes registers

The Instruction set attributes registers are:

- *Instruction Set Attributes Register 0 (ID_ISAR0)* on page 3-25
- *Instruction Set Attributes Register 1 (ID_ISAR1)* on page 3-26
- *Instruction Set Attributes Register 2 (ID_ISAR2)* on page 3-27
- *Instruction Set Attributes Register 3 (ID_ISAR3)* on page 3-28
- *Instruction Set Attributes Register 4 (ID_ISAR4)* on page 3-29.

Feature version registers are read-only registers and are accessed with the following CP15 instructions:

```
MRC p15,0,<Rd>,c0,c2,{0-7} ; reads feature version registers
```

Depending on the Opcode_2 value, the accessed register is:

- CRm=2
  - Opcode_2 == 0: ID_ISAR0, ISA feature Register 0
  - Opcode_2 == 1: ID_ISAR1, ISA Feature Register 1
  - Opcode_2 == 2: ID_ISAR2, ISA Feature Register 2
  - Opcode_2 == 3: ID_ISAR3, ISA Feature Register 3
  - Opcode_2 == 4: ID_ISAR4, ISA Feature Register 4
  - Opcode_2 == 5: RESERVED
  - Opcode_2 == 6: RESERVED
  - Opcode_2 == 7: RESERVED.

RESERVED Opcode_2 combination registers are all read as zero.

Instruction Set Attributes Register 0 (ID_ISAR0)

The purpose of the Instruction Set Attributes Register 0 is to provide information about the instruction set that the processor supports beyond the basic set.

The Instruction Set Attributes Register 0 is:

- in CP15 c0
- a 32-bit read-only register
- accessible in privileged modes only.

Figure 3-13 shows the bit arrangement for the Instruction Set Attributes Register 0.

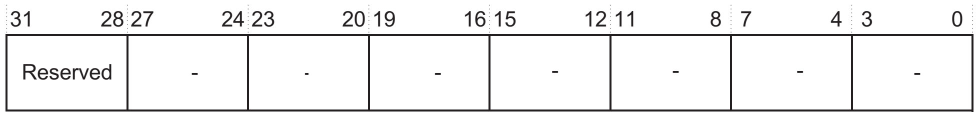


Figure 3-13 Instruction Set Attributes Register o format

Table 3-13 describes the meaning of the ID_ISAR0 bits.

Table 3-13 Instruction Set Attributes Register 0 bit assignments

Bit	Name	Value
[31:28]	RESERVED	RAZ
[27:24]	Divide instructions	`0x0` (not supported)
[23:20]	Debug instructions	`0x1` (BKPT)
[19:16]	Coprocessor instructions	`0x0` (not supported - other than separately attributed architectures)
[15:12]	CmpBranch instructions	`0x0` (not supported)
[11:8]	Bitfield_instructions	`0x0` (not supported)
[7:4]	BitCount_instructions	`0x1` (CLZ)
[3:0]	Atomic instructions	`0x1` (SWP, SWPB)

Instruction Set Attributes Register 1 (ID_ISAR1)

The purpose of the Instruction Set Attributes Register 1 is to provide information about the instruction set that the processor supports beyond the basic set.

The Instruction Set Attributes Register 1 is:

- in CP15 c0
- a 32-bit read-only register
- accessible in privileged modes only.

Figure 3-14 shows the bit arrangement for Instruction Set Attributes Register 1.

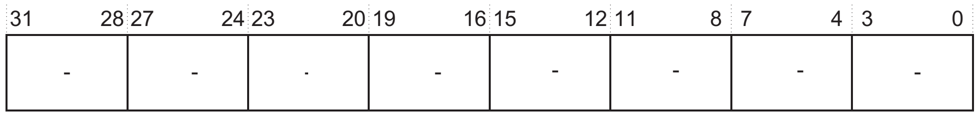


Figure 3-14 Instruction Set Attributes Register 1 format

Table 3-14 describes the meaning of the ID_SAR1 bits.

Table 3-14 Instruction Set Attributes Register 1 bit assignments

Bit	Name	Value
[31:28]	Jazelle instructions	`0x1` (BXJ and J bit in PSRs)
[27:24]	Interwork instructions	`0x2` (BX, BLX, T bit in PSRs and PC loads have BX-like behavior)
[23:20]	Immediate instructions	`0x0` (no special immediate-generating instructions)
[19:16]	IfThen instructions	`0x0` (not supported)
[15:12]	Extend instructions	`0x2` (all supported)
[11:8]	Exception2 instructions	`0x1` (SRS, RFE, CPS)
[7:4]	Exception1 instructions	`0x1` (LDM(2), LDM(3), STM(2))
[3:0]	Endian instructions	`0x1` (SETEND)

Instruction Set Attributes Register 2 (ID_ISAR2)

The purpose of the Instruction Set Attributes Register 2 is to provide information about the instruction set that the processor supports beyond the basic set.

The Instruction Set Attributes Register 2 is:

- in CP15 c0
- a 32-bit read-only register
- accessible in privileged modes only.

Figure 3-15 shows the bit arrangement for Instruction Set Attributes Register 2.

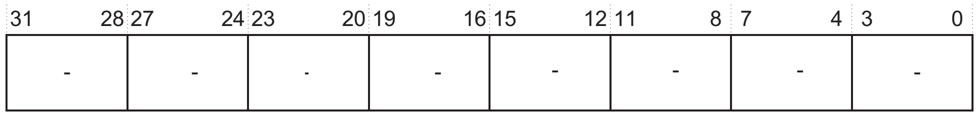


Figure 3-15 Instruction Set Attributes Register 2 format

Table 3-15 describes the meaning of the IDSAR2 bits.

Table 3-15 Instruction Set Attributes Register 2 bit assignments

Bit	Name	Value
[31:28]	Reversal instructions	`0x1` (REV, REV16, REVSH)
[27:24]	PSR instructions	`0x1` (MRS, MSR and "exception return" data-processing instructions
[23:20]	Multiply instructions (advanced, unsigned)	`0x2` (UMULL, UMLAL, UMAAL)
[19:16]	Multiply instructions (advanced, signed)	`0x2` (SMULL, SMAL, SMLABB, SMLABT, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, SMULWT, and Q flag in PSRs)
[15:12]	Multiply instructions	`0x1` (MUL, MLA)
[11:8]	Multi-access interruptible instructions	`0x0` (non-interruptible)
[7:4]	MemoryHint instructions	`0x1` (PLD)
[3:0]	LoadStore instructions	`0x1` (adds LDRD/STRD)

Instruction Set Attributes Register 3 (ID_ISAR3)

The purpose of the Instruction Set Attributes Register 3 is to provide information about the instruction set that the processor supports beyond the basic set.

The Instruction Set Attributes Register 3 is:

- in CP15 c0
- a 32-bit read-only registers
- accessible in privileged modes only.

Figure 3-16 shows the bit arrangement for Instruction Set Attributes Register 3.

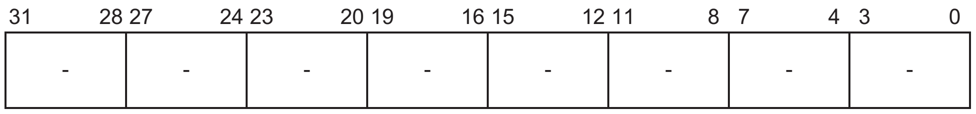


Figure 3-16 Instruction Set Attributes Register 3 format

Table 3-16 describes the meaning of the ID_SAR3 bits.

Table 3-16 Instruction Set Attributes Register 3 bit assignments

Bit	Name	Value
[31:28]	Thumb-2 executable environment Extension instructions	`0x0` (not supported)
[27:24]	TrueNOP instructions	`0x1` (NOP32)
[23:20]	ThumbCopy instructions	`0x1` (Thumb MOV(3) low reg -> low reg and CPY alias)
[19:16]	TableBranch instructions	`0x0` (not supported)
[15:12]	SyncPrim instructions	`0x2` (LDREX, STREX, LDRBEX, STRBEX, LDRHEX, STRHEX, LDRDEX, STRDEX, CLREX)
[11:8]	SWI instructions	`0x1` (supported)
[7:4]	SIMD instructions	`0x3` (all supported)
[3:0]	Saturate instructions	`0x1` (QADD, QDADD, QDSUB, QSUB, and Q flag in PSRs)

Instruction Set Attributes Register 4 (ID_ISAR4)

The purpose of the Instruction Set Attributes Register 4 is to provide information about the instruction set that the processor supports beyond the basic set.

The Instruction Set Attributes Register 4 is:

- in CP15 c0
- a 32-bit read-only register
- accessible in privileged modes only.

Figure 3-17 on page 3-30 shows the bit arrangement for Instruction Set Attributes Register 4.

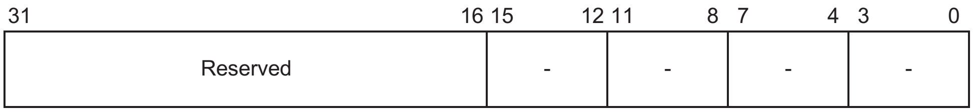


Figure 3-17 Instruction Set Attributes Register 4 format

Table 3-17 describes the meaning of the ISAR4 bits.

Table 3-17 Instruction Set Attributes Register 4 bit assignments

Bit	Name	Value
[31:16]	Reserved	RAZ
[15:12]	SMI instructions	0x0 (not supported)
[11:8]	Writeback instructions	`0x1` (all currently-defined writeback addressing modes supported)
[7:4]	With Shift instructions	`0x4` (all shift options supported)
[3:0]	Unprivileged instructions	`0x1` (LDRBT, LDRT, STRBT, STRT)

3.4.7 c1, Control Register

The purpose of the Control Register is to provide control and configuration of:

- memory alignment, endianness, protection, and fault behavior
- MMU and cache enables
- interrupts
- the location for exception vectors
- program flow prediction.

The Control Register is:

- in CP15 c1
- a 32 bit register,
- accessible in privileged modes only.

You can use the Control Register to enable and disable system configuration options. You can access the Control Register by reading or writing CP15 c1 with the CRm and Opcode_2 fields set to 0:

```
MRC p15,0,<Rd>,c1,c0,0; Read Control Register configuration data
```

```
MCR p15,0,<Rd>,c1,c0,0; Write Control Register configuration data
```

It is recommended that you access this register using a read-modify-write sequence.

All defined control bits are set to zero on Reset except:

- the V bit that is set to zero at Reset if the **VINITHI** signal is LOW, or one if the **VINITHI** signal is HIGH
- The U and EE bits reset values in CP15 Control Register and E Bit reset value in CPSR/SPSR depend on system configuration pins **CFGEND** as described in Table 3-18.

Table 3-18 CFGEND, EE, U, and E bit values

CFGEND [1:0]	CP15 Control Register		CPSR/SPSR
	EE Bit	U Bit	E Bit
00	0	0	0
01 RESERVED	-	-	-
10	0	1	0
11	1	1	1

Table 3-19 shows endianness and alignment control options.

Table 3-19 Endianness and alignment control options

U	A	E	Instruction Endianness	Data Endianness	Unaligned Behavior	Description
0	0	0	LE	LE	Rotated LDR	Legacy LE
0	0	1	-	-	-	Reserved
0	1	0	LE	LE	Data Abort	Modulo 8 LDRD/STRD doubleword alignment checking
0	1	1	LE	BE-8	Data Abort	Modulo 8 LDRD/STRD doubleword alignment checking
1	0	0	LE	LE	Unaligned	Unaligned access permitted
1	0	1	LE	BE-8	Unaligned	Unaligned access permitted
1	1	0	LE	LE	Data Abort	Modulo 4 alignment checking
1	1	1	LE	BE-8	Data Abort	Modulo 4 alignment checking

Figure 3-18 shows the format of the Control Register.

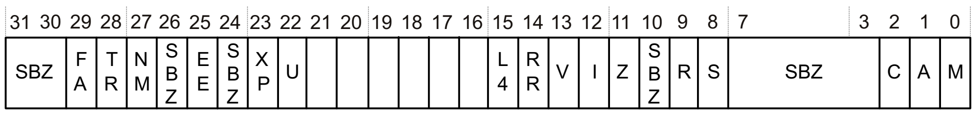


Figure 3-18 Control Register format

Table 3-20 describes the functions of the Control Register bits.

Table 3-20 Control Register bit functions

Bit	Name	Function
[31:30]	-	Reserved. SBZ/RAZ
[29]	Force AP bit	This bit determines how AP bits are interpreted by the TLB. 0 = Access Bit not used 1 = AP[0] used as Access Bit.
[28]	TEX Remap bit	This bit determines how TEX, C and B bits are interpreted by the TLB. 0 = No remapping 1 = Remap registers are used for remapping.

Table 3-20 Control Register bit functions (continued)

Bit	Name	Function
[27]	NMFI bit	NMFI bit 0 = normal FIQ behavior 1 = FIQs behave as NMFIs.
[26]	-	Reserved. SBZ/RAZ
[25]	EE bit	This bit determines the setting of the CPSR E bit on taking an exception: 0 = CPSR E bit is set to 0 on taking an exception 1 = CPSR E bit is set to 1 on taking an exception.
[24]	-	Reserved. SBZ/RAZ
[23]	XP bit	Configure extended page table configuration. This bit configures the hardware page translation mechanism: 0 = Subpage AP bits enabled 1 = Subpage AP bits disabled.
[22]	U bit	This bit enables unaligned data access operation, including support for mixed little-endian and big-endian data.
[21:16]	-	Writing to these bits has no effect. Read as 6'b000101.
[15]	L4 bit	Configure if load instructions to PC set T bit: 0 = Loads to PC set the T bit 1 = Loads to PC do not set the T bit (ARMv4 behavior). For more details see the *ARM Architecture Reference Manual*.
[14]	-	Reserved. SBO/RAO
[13]	V bit	Location of exception vectors: 0 = Normal exception vectors selected, address range = `0x00000000-0x0000001C` 1 = High exception vectors selected, address range = `0xFFFF0000-0xFFFF001C`.
[12]	I bit	Level one Instruction Cache enable/disable: 0 = Instruction Cache disabled 1 = Instruction Cache enabled.

Table 3-20 Control Register bit functions (continued)

Bit	Name	Function
[11]	Z bit	Program flow prediction: 0 = Program flow prediction disabled 1 = Program flow prediction enabled. Program flow prediction includes static and dynamic branch prediction and the return stack. This bit enables all three forms of program flow prediction. You can enable or disable each form individually. See *c1, Auxiliary Control Register* on page 3-34.
[10]	-	SBZ
[9]	R bit	ROM protection. (Deprecated)
[8]	S bit	System protection. (Deprecated)
[7:3]	-	Reserved. SBZ
[2]	C bit	Level one Data Cache enable/disable: 0 = Data cache disabled 1 = Data cache enabled.
[1]	A bit	Strict data address alignment fault enable/disable: 0 = Strict alignment fault checking disabled 1 = Strict alignment fault checking enabled. The A bit setting takes priority over the U bit. The Data Abort trap is taken if strict alignment is enabled and the data access is not aligned to the width of the accessed data item.
[0]	M bit	MMU enable/disable: 0 = MMU disabled 1 = MMU enabled.

Take care with the address mapping of the code sequence used to enable the MMU (see *Enabling the MMU* on page 5-9). See *Disabling the MMU* on page 5-9 for restrictions and effects of having caches enabled with the MMU disabled.

3.4.8 c1, Auxiliary Control Register

The purpose of the Auxiliary Control Register is to control:

- program flow
- coherency mode (SMP/AMP)
- cache exclusive behavior.

The Auxiliary Control Register is:

- in CP15 c1
- a 32-bit read-only register
- accessible in privileged modes only.

You can use the Auxiliary Control Register to enable and disable program flow prediction operations. It is selected by reading or writing CP15 c1 with the Opcode_2 field set to 1:

```
MRC p15,0,<Rd>,c1,c0,1; Read Auxiliary Control Register
```

```
MCR p15,0,<Rd>,c1,c0,1; Write Auxiliary Control Register
```

Figure 3-19 on page 3-35 shows the format of the Auxiliary Control Register.

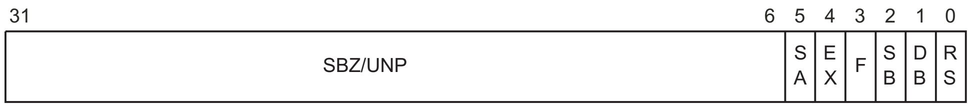


Figure 3-19 Auxiliary Control Register format

Table 3-21 shows the functions of the bit fields in the Auxiliary Control Register.

Table 3-21 Auxiliary Control Register bit functions

Bits	Name	Function
[31:6]	-	Reserved. These bits must be updated using a read-modify-write technique to ensure that currently unallocated bits are not unnecessarily modified.
[5]	SMP / nAMP mode	Signals if the CPU is taking part in coherency or not. See *AMP mode and SMP mode* on page 3-36.
[4]	EXCL	This bit enables the exclusive behavior of L1 if the core is used in conjunction with an L2 cache supporting that behavior: 1 = L1 and L2 caches are exclusive 0 = L1 and L2 caches are inclusive (default).
[3]	F bit	Instruction folding enable. This bit enables the use of instruction folding if program flow prediction is enabled. See CP15, Control Register.

Table 3-21 Auxiliary Control Register bit functions (continued)

Bits	Name	Function
[2]	SB	Static branch prediction enable. This bit enables the use of static branch prediction if program flow prediction is enabled. See CP15, Control Register. 1 = Static branch prediction is enabled. 0 = Static branch prediction is disabled This bit is set on reset.
[1]	DB	Dynamic branch prediction enable. This bit enables the use of dynamic branch prediction if program flow prediction is enabled. See CP15, Control Register. 0 = Dynamic branch prediction is disabled 1 = Dynamic branch prediction is enabled. This bit is set on reset.
[0]	RS	Return stack enable. This bit enables the use of the return stack if program flow prediction is enabled. See CP15, Control Register. 0 = Return stack is disabled 1 = Return stack is enabled. This bit is set on reset.

AMP mode and SMP mode

By default, the ARM11 MPCore processor is in AMP mode (bit 5 reset to 0). To prevent coherent data corruption the sequence to turn on MP11 CPUs in SMP mode is:

1. Write the SCU register to change CPU mode.
2. Disable interrupts.
3. Clean and invalidate all the D-cache.
4. Write SMP/nAMP bit as 1.
5. Enable interrupts.

Similarly, the sequence to turn on MP11 CPUs in AMP mode is:

1. Disable interrupts
2. Clean and invalidate all the D-cache
3. Write SMP/nAMP bit as 0
4. Write the SCU register to change the CPU mode.
5. Enable interrupts.

Note

In AMP mode, shared write-back write-allocate regions are treated as noncachable just like other shared regions.

3.4.9 c1, Coprocessor Access Control Register

The Coprocessor Access Control Register controls accesses to all coprocessors other than CP14 and CP15, that is, VFP because ARM11 MPCore processors do not support generic coprocessors.

The Coprocessor Access Control Register is:

- in CP15 c1
- a 32-bit read/write register
- accessible in privileged modes only.

You can access the Coprocessor Access Control Register by reading or writing CP15 c1 with the Opcode_2 field set to 2:

```
MRC p15,0,<Rd>,c1,c0,2; Read Coprocessor Access Control Register
MCR p15,0,<Rd>,c1,c0,2; Write Coprocessor Access Control Register
```

Figure 3-20 on page 3-37 shows the format of the Coprocessor Access Control Register.

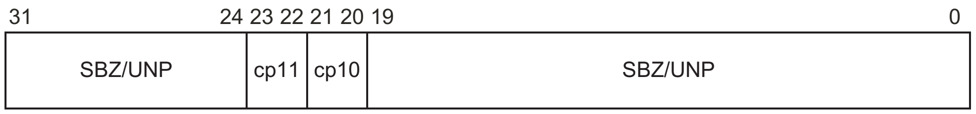


Figure 3-20 Coprocessor Access Control Register format

Table 3-22 shows the bit-pair access rights encoding for each coprocessor connected to the ARM11 MPCore processor.

Table 3-22 Coprocessor access rights

Bits	Meaning
b00	Access denied. Attempts to access the corresponding coprocessor generate an Undefined exception.
b01	Supervisor access only.
b10	Reserved.
b11	Full access.

After updating this register you must execute an *Instruction Memory Barrier* (IMB) sequence. None of the instructions executed after changing this register and before the IMB must be coprocessor instructions affected by the change in coprocessor access rights.

After a system reset, all coprocessor access rights are set to Access denied.

If a coprocessor is not implemented then attempting to write the coprocessor access rights bits for that entry to values other than b00 has no effect. This mechanism can be used by software to determine which coprocessors are present.

3.4.10 c2, Translation Table Base Register 0

The purpose of the Translation Table Base Register 0 is to hold the physical address of the first-level translation table.

The Translation Table Base Register 0 is:

- in CP15 c2
- a 32 bit read/write register
- accessible in privileged modes only.

Use Translation Table Base Register 0 for process-specific addresses, where each process maintains a separate first-level page table. On a context switch you must modify both Translation Table Base Register 0 and the Translation Table Base Control Register, if appropriate.

You can access the Translation Table Base Register 0 by reading or writing CP15 c2 with the Opcode_2 field set to 0:

```
MRC p15, 0, <Rd>, c2, c0, 0; Read Translation Table Base Register 0
```

```
MCR p15, 0, <Rd>, c2, c0, 0; Write Translation Table Base Register 0
```

Figure 3-21 shows the format of the bits in Translation Table Base Register 0. For an explanation of N in the figure see *c2, Translation Table Base Control Register* on page 3-40.

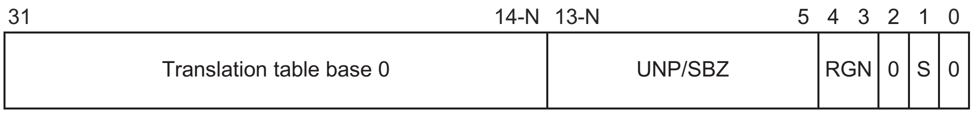


Figure 3-21 Translation Table Base Register 0 format

Table 3-23 shows the functions of the bits in the Translation Table Base Register 0. For an explanation of N in the table see *c2, Translation Table Base Control Register* on page 3-40.

Table 3-23 Translation Table Base Register 0 bits

Bits	Name	Function
[31:14-N]	Translation table base 0	Pointer to the level one translation table
[13-N:5]	-	UNP/SBZ

[4:3]	RGN	Outer cachable attributes for page table walking:
		b00 = Outer Noncachable
		b01 = Outer Cachable Write-Back cached, Write Allocate
		b10 = Outer Cachable Write-Through, No Allocate on Write
		b11 = Outer Cachable Write-Back, No Allocate on Write

[2]	-	UNP/SBZ
[1]	S	The page table walk is to Shared (1) or Non-Shared (0) memory.
[0]	-	UNP/SBZ

3.4.11 c2, Translation Table Base Register 1

The purpose of the Translation Table Base Register 1 is to hold the physical address of the first-level table. The expected use of the Translation Table Base Register 1 is for OS and I/O addresses.

The Translation Table Base Register 1 is:

- in CP15 c2
- a 32 bit read/write register
- accessible in privileged modes only.

You can access Translation Table Base Register 1 by reading or writing CP15 c2 with the Opcode_2 field set to 0:

```
MRC p15, 0, <Rd>, c2, c0, 1 Read Translation Table Base Register 1
```

```
MCR p15, 0, <Rd>, c2, c0, 1; Write Translation Table Base Register 1
```

Figure 3-22 shows the format of the bits in Translation Table Base Register 1.

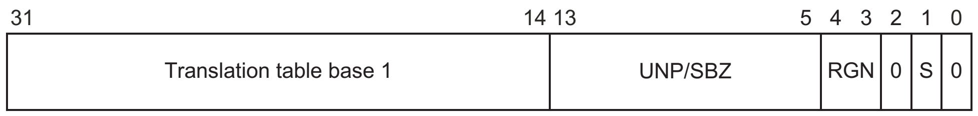


Figure 3-22 Translation Table Base Register 1 format

Writing to CP15 c2 updates the pointer to the first-level translation table from the value in bits [31:14] of the written value. Bits [13:5] Should Be Zero. Translation Table Base Register 1 must reside on a 16KB page boundary.

Table 3-24 shows the functions of the bits in the Translation Table Base Register 1.

Table 3-24 Translation Table Base Register 1 bits

Bits	Name	Function
[31:14]	Translation table base 1	Pointer to the level one translation table.
[13:5]	-	UNP/SBZ

[4:3]	RGN	Outer cachable attributes for page table walking:
		b00 = Outer Noncachable
		b01 = Outer Cachable Write-Back cached, Write Allocate
		b10 = Outer Cachable Write-Through, No Allocate on Write
		b11 = Outer Cachable Write-Back, No Allocate on Write.

[2]	P	Indicates to the memory controller that, if supported ECC is (1) enabled or (0) disabled. For ARM11 MPCore processors this bit Should Be Zero.
[1]	S	The page table walk is to Shared (1) or Non-Shared (0) memory.
[0]	C	The page table walk is Inner Cachable (1) or Inner Noncachable (0). For ARM11 MPCore processors this bit is 0.

3.4.12 c2, Translation Table Base Control Register

The purpose of the Translation Table Base Control Register is to determine if a page table miss for a specific VA uses, for its page table walk, either:

- Translation Table Base Register 0. The recommended use is for task-specific addresses
- Translation Table Base Register 1. The recommended use is for operating system and I/O addresses.

The Translation Table Base Control Register is:

- in CP15 c2
- a 32 bit read/write register
- accessible in privileged modes only.

You can access the Translation Table Base Control Register by reading or writing CP15 c2 with the Opcode_2 field set to 2:

```
MRC p15, 0, <Rd>, c2, c0, 2 ; Read Translation Table Base Control Register
```

```
MCR p15, 0, <Rd>, c2, c0, 2 ; Write Translation Table Base Control Register
```

Figure 3-23 shows the format of the bits in the Translation Table Base Control Register.

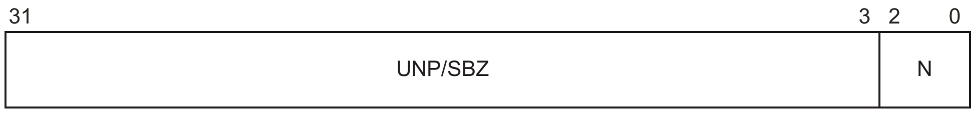


Figure 3-23 Translation Table Base Control Register format

The page table base register is selected as follows:

1. If N = 0, always use Translation Table Base Register 0. This is the default case at reset. It is backwards compatible with ARMv5 or earlier processors.
2. If N is greater than 0, then if bits [31:32-N] of the Virtual Address are all 0, use Translation Table Base Register 0, otherwise use Translation Table Base Register 1. N must be in the range 0-7.

Reading from CP15 c2 returns the size of the page table boundary for Translation Table Base Register 0. Bits [31:3] Should Be Zero.

Writing to CP15 c2 updates the size of the first-level translation table base boundary for Translation Table Base Register 0. Bits [31:3] Should Be Zero.

Table 3-25 shows the values of N for Translation Table Base Register 0.

Table 3-25 Values of N for Translation Table Base Register 0

N	Translation Table Base Register 0 page table boundary size
0	16KB
1	8KB
2	4KB
3	2KB
4	1KB
5	512-byte
6	256-byte
7	128-byte

3.4.13 c3, Domain Access Control Register

The purpose of the Domain Access Control Register is to hold the access permissions for a maximum of 16 domains.

The Domain Access Control Register is:

- in CP15 c3
- a 32-bit read/write register
- accessible in privileged modes only.

You can access the Domain Access Control Register by reading or writing CP15 c3 with the CRm and Opcode_2 fields set to 0:

```
MRC p15, 0, <Rd>, c3, c0, 0; Read Domain Access Control Register
```

```
MCR p15, 0, <Rd>, c3, c0, 0; Write Domain Access Control Register
```

Figure 3-24 shows the two-bit domain access permission fields of the Domain Access Control Register.

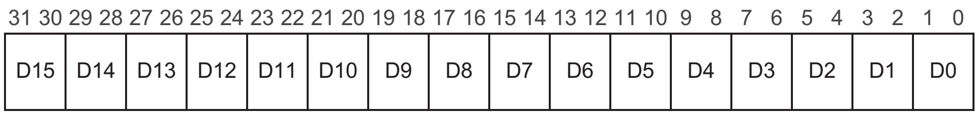


Figure 3-24 Domain Access Control Register format

Table 3-26 shows the encoding of the bits in the Domain Access Control Register.

Table 3-26 Encoding of domain bits in CP15 c3

Value	Access type	Description
b00	No access	Any access generates a domain fault
b01	Client	Accesses are checked against the access permission bits in the TLB entry
b10	Reserved	Any access generates a domain fault
b11	Manager	Accesses are not checked against the access permission bits in the TLB entry, so a permission fault cannot be generated

3.4.14 c5, Data Fault Status Register

The purpose of the Data Fault Status Register is to hold the source of the last data fault. The Data Fault Status Register indicates the domain and type of access being attempted when an abort occurred.

The Data Fault Status Register is:

- in CP15 c5
- a 32-bit read/write register
- accessible in privileged modes only.

You can access the Data Fault Status Register by reading or writing CP15 c5 with the CRm and Opcode_2 fields set to 0:

```
MRC p15, 0, <Rd>, c5, c0, 0; Read Data Fault Status Register
```

```
MCR p15, 0, <Rd>, c5, c0, 0; Write Data Fault Status Register
```

Figure 3-25 shows the format of the Data Fault Status Register.

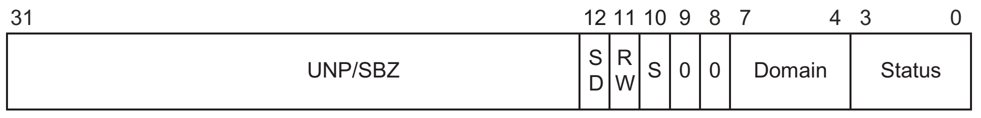


Figure 3-25 Data Fault Status Register format

Table 3-27 shows the bit fields for the Data Fault Status Register.

Table 3-27 Data Fault Status Register bits

Bits	Meaning
[31:13]	UNP/SBZ.
[12]	SD External Abort Qualifier 1= External Abort marked as SLVERR 0=External Abort marked as DECERR[a]
[11]	Not Read/Write. Indicates what type of access caused the abort: 0 = Read 1 = Write Aborts on CP15 operations. This bit is set to 1.
[10]	Part of the Status field. See Bits [3:0] in this table.
[9:8]	Always read as 0.
[7:4]	Specifies which of the 16 domains (D15-D0) was being accessed when a data fault occurred.
[3:0]	Type of fault generated (see *Fault status and address* on page 5-38).

a. SLVERR and DECERR are the two possible types of abort reported in an AXI bus.

See *Fault status and address* on page 5-38 for the encodings of the DFSR bits.

Reading CP15 c5 with Opcode_2 set to 0 returns the value of the Data Fault Status Register.

Writing CP15 c5 with Opcode_2 set to 0 sets the Data Fault Status Register to the value of the data written. This is useful for a debugger to restore the value of the Data Fault Status Register. The register must be written using a read-modify-write sequence.

3.4.15 c5, Instruction Fault Status Register

The purpose of the *Instruction Fault Status Register* (IFSR) is to hold the source of the last instruction fault. The IFSR indicates the type of access being attempted when an abort occurred.

The Instruction Fault Status Register is:

- in CP15 c5
- a 32-bit read/write register
- accessible in privileged modes only.

You can access the IFSR by reading or writing CP15 c5 with the Opcode_2 field set to 1:

```
MRC p15, 0, <Rd>, c5, c0, 1; Read Instruction Fault Status Register
```

```
MCR p15, 0, <Rd>, c5, c0, 1; Write Instruction Fault Status Register
```

Figure 3-26 shows the format of the Instruction Fault Status Register.

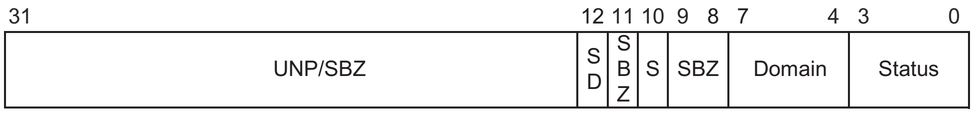


Figure 3-26 Instruction Fault Status Register format

Table 3-28 shows the bit fields for the Instruction Fault Status Register.

Table 3-28 Instruction Fault Status Register bits

Bits	Name	Meaning
[31:13]	-	UNP/SBZ
[12]	SD	External abort qualifier 1 = External abort marked as SLVERR 0 = External abort marked as DECERR
[11]	-	Always reads as 0
[10]	S	Part of the status field
[9:8]	-	Always reads as 0
[7:4]	Domain	Specifies which of the 16 domains (D15-D0) was being accessed when a data fault occurred.
[3:0]	Status	Type of fault generated (see *Fault status and address* on page 5-38)

See *Fault status and address* on page 5-38 for the encoding of the IFSR bits

Reading CP15 c5 with the Opcode_2 field set to 1 returns the value of the IFSR.

Writing CP15 c5 with the Opcode_2 field set to 1 sets the IFSR to the value of the data written. This is useful for a debugger to restore the value of the IFSR. The register must be written using a read-modify-write sequence. Bits [31:4] Should Be Zero.

3.4.16 c6, Fault Address Register

The purpose of the *Fault Address Register* (FAR) is to hold the *Modified Virtual Address* (MVA) of the fault when a precise fault occurs. The FAR is only updated for precise data faults, not for imprecise data faults or prefetch faults.

The FAR is:

- in CP15 c6
- a 32-bit read/write register
- accessible in privileged modes only.

Writing CP15 c6 with Opcode_2 set to 0 sets a FAR to the value of the data written. This is useful for a debugger to restore the value of a FAR.

The ARM11 MPCore processor also updates the FAR on debug exception entry because of watchpoints. This is architecturally Unpredictable. See *Effect of a debug event on CP15 registers* on page 12-29 for more details.

3.4.17 c6, Watchpoint Fault Address Register

Reading CP15 c6 returns the *Watchpoint Fault Address Register* (WFAR) as specified by the Opcode_2 value.

You can access the Watchpoint Fault Address Register by reading or writing CP15 c6 with the Opcode_2 field set to 1:

```
MRC p15, 0, <Rd>, c6, c0, 1; Read Watchpoint Fault Address Register
```

```
MCR p15, 0, <Rd>, c6, c0, 1; Write Watchpoint Fault Address Register
```

The WFAR holds the Virtual Address of the instruction that triggered the watchpoint. The contents are Unpredictable after a precise Data Abort or Instruction Abort occurs.

If the watchpoint is taken when in ARM state, the WFAR contains the address of the instruction that triggered it plus `0x8`. If the watchpoint is taken while in Thumb state, the WFAR contains the address of the instruction that triggered it plus `0x4`. If the watchpoint is taken while in Java state, the WFAR contains the address of the instruction causing it.

Writing CP15 c6 with Opcode_2 set to 1 sets the WFAR to the value of the data written. This is useful for a debugger to restore the value of the WFAR.

3.4.18 c7, Cache Operations Register

The purpose of c7 is to:

- control these operations:
  - clean and invalidate instruction and data caches
  - flush prefetch buffer
  - flush branch target address cache
  - virtual to physical address translation
- implement the *Data Write Barrier* (DWB) operation
- implement the *Data Memory Barrier* (DMB) operation
- implement the Wait For Interrupt clock control function.

You can use c7 to control the instruction and data caches. You can also use it to implement similar functions on prefetch buffers and branch target caches.

You can use the following instruction to write to CP15 c7:

```
MCR p15,0, <Rd>, c7, <CRm>, <Opcode_2>
```

The function of each cache operation is selected by the Opcode_2 and CRm fields in the MCR instruction used to write CP15 c7.

Table 3-29 on page 3-49 shows the functions that you can perform using CP15 c7.

Writing c7 with a combination of CRm and Opcode_2 not listed in Table 3-29 on page 3-49 gives Unpredictable results.

If Opcode_1 = 0, these instructions are applied to a level one cache system. All other Opcode_1 values are reserved.

All CP15 c7 operations can only be executed in a privileged mode of operation, except Data Synchronization Barrier, Flush Prefetch Buffer, and Data Memory Barrier. These can be operated in User mode. Attempting to execute a privileged instruction in User mode results in the Undefined instruction trap being taken.

The following definitions apply to Table 3-29 on page 3-49:

Wait for interrupt

Puts the ARM into a low power state and stops it executing further until an interrupt, or a debug request, occurs. Interrupt and debug events always cause the ARM processor to restart, irrespective of whether the interrupt is masked. Debug events require debug enabled.

When an interrupt does occur, the MCR instruction completes and either the next instruction executes (if an interrupt event and the interrupt is masked), or the IRQ or FIQ handler is entered as normal. The return link in R14_irq or R14_fiq contains the address of the MCR instruction plus 8, so that the normal instruction used for interrupt return (`SUBS PC,R14, #4`) returns to the instruction following the MCR.

Clean Applies to write-back data caches. This means that if the cache line contains stored data that has not yet been written out to main memory, it is written to main memory now, and the line is marked as clean.

Invalidate This means that the cache line (or all the lines in the cache) is marked as invalid, so that no cache hits occur for that line until it is re-allocated to an address. For write-back data caches, this does not include cleaning the cache line unless that is also stated.

Data Synchronization Barrier

The DataSynchronizationBarrier operation acts as a special kind of memory barrier. The DSB operation completes when:

- All explicit reads and writes before this instruction complete.
- All Cache, Branch predictor and TLB maintenance operations preceding this instruction complete.

In addition, no instruction subsequent to the DSB may execute until the DSB completes.

Data Memory Barrier DataMemoryBarrier (DMB)

DMB acts as a general memory barrier, exhibiting the following behavior:

- All explicit memory accesses by instructions occurring in program order before this instruction are globally observed before any memory accesses due to instructions occurring in program order after this instruction are observed.
- DataMemoryBarrier has no effect on the ordering of other instructions executing on the processor.

As such, DMB ensures the apparent order of the explicit memory operations before and after the instruction, without ensuring their completion.

Flush Prefetch Buffer

Flushing the instruction prefetch buffer has the effect that all instructions occurring in program order after this instruction are fetched from the memory system after the execution of this instruction, including the level one cache. This operation is useful for ensuring the correct execution of self-modifying code. See *Explicit memory barriers* on page 5-28.

Data This is the value that is written to CP15 c7. This is the value in the Rd register specified in the MCR instruction.

If the data is stated to be a Virtual Address, it does not have to be cache line aligned. This address is looked up in the cache for the particular operation. Invalidation and cleaning operations have no effect if they miss in the cache. If the corresponding entry is not in the TLB, these instructions can cause a TLB miss exception or hardware page table walk, depending on the miss handling mechanism.

For the cache control operations, the Virtual Addresses that are passed to the cache are not translated by the FCSE extension.

If the data is stated to be Set/Index format (see Figure 3-27 on page 3-50), it identifies the cache line that the operation is to be applied to by specifying which cache Set it belongs to and what its Index is within the Set. The Index corresponds to the number of the cache way, and the Set number corresponds to the line number within a cache way.

Table 3-29 shows the cache operation functions and the associated data and instruction formats for CP15 c7.

Table 3-29 Cache operation functions

Function	Data	Instruction
Wait For Interrupt	SBZ	`MCR p15, 0, <Rd>, c7, c0, 4`
Invalidate Entire Instruction Cache. Also flushes the branch target cache	SBZ	`MCR p15, 0, <Rd>, c7, c5, 0`
Invalidate Instruction Cache Line (using MVA)	MVA	`MCR p15, 0, <Rd>, c7, c5, 1`
Invalidate Instruction Cache Line (using Index)	Set/Index	`MCR p15, 0, <Rd>, c7, c5, 2`
Flush Prefetch Buffer[a]	SBZ	`MCR p15, 0, <Rd>, c7, c5, 4`
Flush Entire Branch Target Cache	SBZ	`MCR p15, 0, <Rd>, c7, c5, 6`

Table 3-29 Cache operation functions (continued)

Function	Data	Instruction
Flush Branch Target Cache Entry	MVA[b]	`MCR p15, 0, <Rd>, c7, c5, 7`
Invalidate Entire Data Cache	SBZ	`MCR p15, 0, <Rd>, c7, c6, 0`
Invalidate Data Cache Line (using MVA)	MVA	`MCR p15, 0, <Rd>, c7, c6, 1`
Invalidate Data Cache Line (using Index)	Set/Index	`MCR p15, 0, <Rd>, c7, c6, 2`
Invalidate Both Caches. Also flushes the branch target cache	SBZ	`MCR p15, 0, <Rd>, c7, c7, 0`
Clean Entire Data Cache	SBZ	`MCR p15, 0, <Rd>, c7, c10, 0`
Clean Data Cache Line (using MVA)	MVA	`MCR p15, 0, <Rd>, c7, c10, 1`
Clean Data Cache Line (using Index)	Set/Index	`MCR p15, 0, <Rd>, c7, c10, 2`
Drain Synchronization Barrier[a]	SBZ	`MCR p15, 0, <Rd>, c7, c10, 4`
Data Memory Barrier[a]	SBZ	`MCR p15, 0, <Rd>, c7, c10, 5`
Clean and Invalidate Entire Data Cache	SBZ	`MCR p15, 0, <Rd>, c7, c14, 0`
Clean and Invalidate Data Cache Line (using MVA)	MVA	`MCR p15, 0, <Rd>, c7, c14, 1`
Clean and Invalidate Data Cache Line (using Index)	Set/Index	`MCR p15, 0, <Rd>, c7, c14, 2`

a. These operations are accessible in both User and privileged modes of operation. All other operations are only accessible in privileged modes of operation.
b. The range of MVA bits used in this function is different to the range of bits used in other functions that have MVA data.

The cache invalidation operations apply to all cache locations, including those locked in the cache. An explicit flush of the relevant lines in the branch target cache must be performed after invalidation of Instruction Cache lines or the results are Unpredictable. This is not required after an entire Instruction Cache invalidation.

The operations that act on a single cache line identify the line using the contents of Rd as the address, passed in the MCR instruction. The data is interpreted using:

- *Set/Index format*
- *Modified Virtual Address (MVA) format* on page 3-51.

Set/Index format

Figure 3-27 shows the Index tag format you can use to specify a line in the cache that must be accessed.

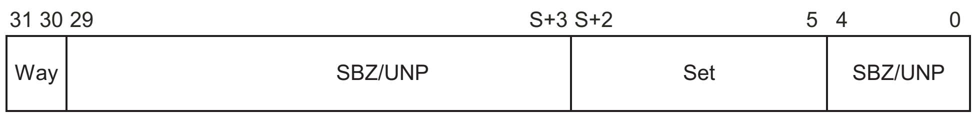


Figure 3-27 Register 7 Set/Index format

Table 3-30 shows the bit fields for Index operations using CP15 c7, and their meanings.

Table 3-30 Bit fields for Set/Index operations using CP15 c7

Bits	Name	Description
[31:30]	Way	Way in set being accessed
[29:S+3]	-	SBZ/UNP
[S+2:5]	Set	Set being accessed
[4:0]	-	SBZ/UNP

The value of S in Table 3-30 is dependent on the cache size. Table 3-31 shows the relationship of cache sizes and the S parameter value.

Table 3-31 Cache size and S parameter dependency

Cache size	S parameter value
16KB	7
32KB	8
64KB	9

The value of S is derived from the following equation:

$$S = \log_2 \left(\frac{\text{Cache size}}{\text{Associativity x line length in bytes}} \right)$$

See *c0, TLB Type Register* on page 3-14 for details in instruction and data cache size.

Example 3-1 is an example using the command Clean Data Cache Line (using Index).

Example 3-1 Clean Data Cache Line (using Index)

```
    ;code is specific to ARM11 MPCore processors with 32KB caches
    MOV R0, #0:SHL:5
seg_loop
    MOV R1, #0:SHL:26
line_loop
    ORR R2,R1,R0
    MCR p15,0,R2,c7,c10,2
    ADD R1,R1,#1:SHL:26
    CMP R1,#0
    BNE line_loop
    ADD R0,R0,#1:SHL:5
    CMP R0,#1:SHL:9
    BNE seg_loop
```

Modified Virtual Address (MVA) format

The MVA format is useful for flushing a particular address or range of addresses in the caches. Figure 3-28 shows the MVA format for c7 functions:

- Invalidate Instruction Cache Line
- Invalidate Data Cache Line
- Clean Data Cache Line
- Prefetch Instruction Cache Line
- Clean and Invalidate Data Cache Line.

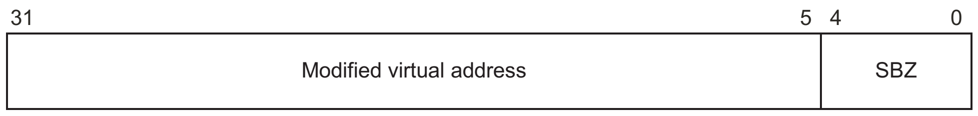


Figure 3-28 CP15 Register c7 MVA format

Bits [4:0] are ignored.

Figure 3-29 shows the MVA format for c7 Flush Branch Target Cache Entry function.

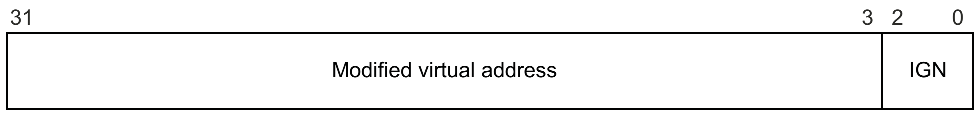


Figure 3-29 CP15 c7 MVA format for Flush Branch Target Cache Entry function

Bits [2:0] are ignored.

Invalidate, Clean, and Clean and Invalidate, Entire Data Cache operations

CP15 c7 specifies operations for cleaning the entire Data Cache, and also for performing a clean and invalidate of the entire Data Cache. These are blocking operations that can be interrupted. If they are interrupted, the r14 value that is captured on the interrupt is the address of the instruction that launched the cache operation + 4. This enables the standard return mechanism for interrupts to restart the operation.

All operations on entire Data and or Instruction caches are interruptible and restartable. When interrupted, these operations stop and automatically restart from where they were interrupted.

User access to CP15 c7 operations

A small number of CP15 c7 operations can be executed by code while in User mode. Attempting to execute a privileged operation in User mode using CP15 c7 results in an Undefined instruction trap being taken.

3.4.19 c7, VA to PA operations

This section describes VA to PA operations:

- *VA to PA Translation Register*
- *PA Register* on page 3-53.

VA to PA Translation Register

A write to the VA to PA Translation Register translates the true virtual address (not the MVA) provided by a general-purpose register (<Rn> Field) and stores the corresponding physical address in the PA Register. Figure 3-30 on page 3-53 shows the register format.

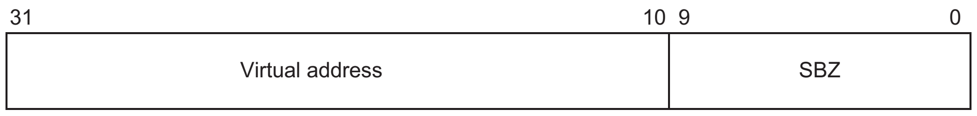


Figure 3-30 VA to PA register format

The VA to PA translation can only be performed in privileged mode and uses the current ASID (in the Context ID Register) to perform the comparison in the TLB.

The VA to PA Translation Register is accessed by writing to CP15 c7 register with the <CRm> field set to c8 and the Opcode_2 field being used to select which kind of permission check will be performed during the translation:

```
MCR p15,0,<Rn>,c7,c8,0; VA to PA translation with privileged read permission check
MCR p15,0,<Rn>,c7,c8,1; VA to PA translation with privileged write permission check
MCR p15,0,<Rn>,c7,c8,2; VA to PA translation with user read permission check
MCR p15,0,<Rn>,c7,c8,3; VA to PA translation with user write permission check.
```

PA Register

The purpose of the PA Register is to hold:

- the PA after a successful translation
- the source of the abort for an unsuccessful translation.

The PA Register is:

- in CP15 c7
- a 32 bit read/write register
- accessible in privileged modes only.

The PA Register format depends on the value of bit 0, which signals whether or not there has been an error during the VA to PA translation.

The PA Register is accessed by reading to CP15 c7 with <CRm> field set to c4 and Opcode_2 field set to 0:

```
MRC p15,0,<Rd>,c7,c4,0; Read PA register
```

If the translation has aborted, bits[5:1] give the encoding of the source of the abort as shown in Figure 3-31 on page 3-54. See *c5, Data Fault Status Register* on page 3-42 and *c5, Instruction Fault Status Register* on page 3-44 for information on the Fault Status Register bits and the SD bit.

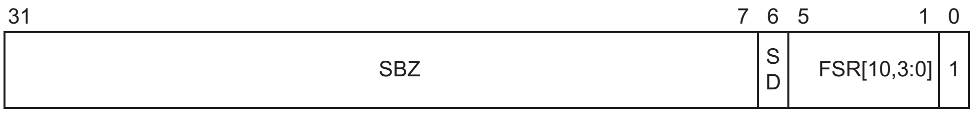


Figure 3-31 PA Register aborted translation

If the translation has completed successfully, PA register format is as shown in Figure 3-32.

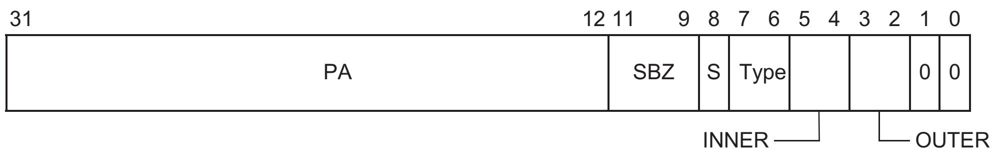


Figure 3-32 PA Register successful translation

Table 4 48 shows the functions of the bits in the PA register.

Table 3-32 PA Register bit assignments

Bit	Name	Function
[31:12]	PA	Physical address
[11:9]	-	SBZ/RAZ
[8]	S Bit	Shareable attribute

[7:6]	Type	Memory Type:
		b00 = Strongly ordered
		b01 = Device memory
		b10 = Normal memory
		b11 = RESERVED

[5:4]	Inner	Signals region inner attributes for normal memory type (type = b10): b11 = Inner write-back. No write-allocate (treated as write allocate) b10 = Inner write-through. No write-allocate (treated as inner noncachable) b01 = Inner Write-back write-allocate b00 = Inner noncachable

[3:2]	Outer	Signals region outer attributes for normal memory type (type = b10):
		b11 = Outer write-back. No write-allocate
		b10 = Outer write-through. No write-allocate
		b01 = Outer Write-back write-allocate
		b00 = Outer noncachable

[1:0]	-	SBZ/RAZ

3.4.20 c8, TLB Operations Register

The purpose of the TLB Operations Register is to either:

- invalidate all the unlocked entries in the TLB
- invalidate all TLB entries for an area of memory before the MMU remaps it

- invalidate all TLB entries that match an ASID value.

The TLB Operations Register is:

- in CP15 c8
- a 32-bit write-only register
- accessible in privileged modes only.

The TLB Operations Register, CP15 c8, is a write-only register used to manage the *Translation Lookaside Buffer* (TLB).

The defined TLB operations are listed in Table 3-33 on page 3-56. The function to be performed is selected by the Opcode_2 and CRm fields in the MCR instruction used to write CP15 c8. Writing other Opcode_2 or CRm values is Unpredictable.

Reading from CP15 c8 is Unpredictable.

Table 3-33 shows the TLB Operations Register instructions.

Table 3-33 TLB Operations Register instructions

Function	Data	Instruction
Invalidate TLB	SBZ	`MCR p15,0,<Rd>,c8,<CRm>,0`
Invalidate TLB Single Entry by MVA with ASID match	MVA/ASID	`MCR p15,0,<Rd>,c8,<CRm>,1`
Invalidate TLB Entries on ASID Match	ASID	`MCR p15,0,<Rd>,c8,<CRm>,2`
Invalidate TLB Single Entry on MVA only	MVA	`MCR p15,0,<Rd>,c8,<CRm>,3`

The CRm value indicates to the hardware what type of access caused the TLB function to be invoked.

Table 3-34 shows the CRm values for the TLB Operations Register, and their meanings. All other CRm values are reserved

Table 3-34 CRm values for TLB Operations Register

CRm	Meaning
c5	Instruction TLB operation
c6	Data TLB operation
c7	Unified TLB operation

Note

The ARM11 MPCore processor has a unified TLB. Any TLB operations specified for the Instruction or Data TLB perform the equivalent operation on the unified TLB.

The Invalidate TLB Single Entry operation uses the Virtual Address as an argument. Figure 3-33 shows the format of this.

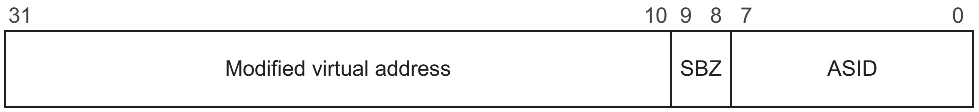


Figure 3-33 TLB Operations Register Virtual Address format

The Invalidate ASID Entries on ASID Match function requires an ASID as an argument. Figure 3-34 shows the format of this.

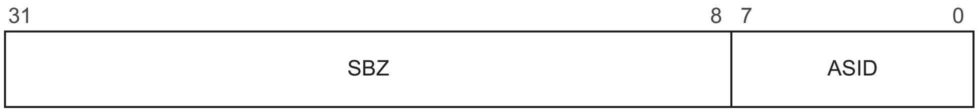


Figure 3-34 TLB Operations Register ASID format

Functions that update the contents of the TLB occur in program order. Therefore, an explicit data access prior to the TLB function uses the old TLB contents, and an explicit data access after the TLB function uses the new TLB contents. For instruction accesses, TLB updates are guaranteed to have taken effect before the next pipeline flush. This includes flush prefetch buffer operations and exception return sequences.

Invalidate TLB

Invalidate TLB invalidates all the unlocked entries in the TLB.

Invalidate TLB Single Entry

You can use Invalidate TLB Single Entry to invalidate an area of memory prior to remapping. You must perform an Invalidate TLB Single Entry of a Virtual Address (VA) in each area to be remapped (section, small page, or large page).

This function invalidates a TLB entry that matches the provided VA and ASID, or a global TLB entry that matches the provided VA. This function invalidates a matching locked entry.

Invalidate TLB Entries on ASID Match

This is a single interruptible operation that invalidates all TLB entries that match the provided ASID value. This function invalidates locked entries. Entries marked as global are not invalidated by this function.

In ARM11 MPCore processors, this operation takes several cycles to complete and the instruction is interruptible. When interrupted the r14 state is set to indicate that the MCR instruction has not executed. Therefore, r14 points to the address of the MCR + 4. The interrupt routine then automatically restarts at the MCR instruction.

If this operation is interrupted and later restarted, any entries fetched into the TLB by the interrupt that uses the provided ASID are invalidated by the restarted invalidation.

Invalidate TLB entries on MVA only

You can use Invalidate TLB Entries to invalidate an area of memory prior to remapping. You must perform an Invalidate TLB Single Entry of a Virtual Address (VA) in each area to be remapped (section, small page, or large page).

This function invalidates a TLB entry that matches the provided VA. This entry can be global or nonglobal. If the entry is nonglobal the ASID matching is ignored. This function invalidates a matching locked entry.

3.4.21 c9, Data Cache Lockdown Register

The purpose of the Data Cache Lockdown Register is to provide a means to lock down the cache and therefore provide some control over pollution that applications might cause. With these registers you can lock down each cache way independently.

The Data Cache Lockdown Register is:

- in CP15 c9
- a 32-bit read/write register
- accessible in privileged modes only.

You can access the Data Cache Lockdown Register by reading or writing CP15 c9 with the CRm field set to c0 and the Opcode_2 field set to 0. For example:

```
MRC p15, 0, <Rd>, c9, c0, 0 ; Read Data Cache Lockdown Register
MCR p15, 0, <Rd>, c9, c0, 0 ; Write Data Cache Lockdown Register
```

ARM11 MPCore processors only support one method of using cache lockdown registers, called Format C. This method is a cache way based locking scheme. It enables you to lockdown each cache way independently. This gives you some control over cache pollution caused by particular applications, in addition to providing a traditional lockdown function for locking critical regions into the cache.

A locking bit for each cache way determines if the cache allocation mechanism is able to access that cache way.

ARM11 MPCore processors have an associativity of 4. If all ways are locked, the ARM11 MPCore processor behaves as if only ways 3 to 1 are locked and way 0 is unlocked.

Figure 3-35 on page 3-59 shows the format of the Data Cache Lockdown Register.

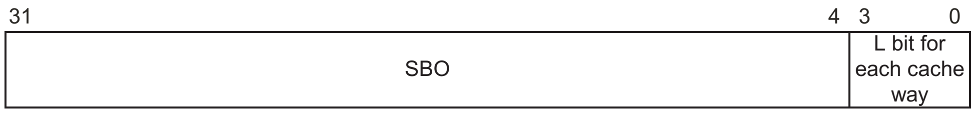


Figure 3-35 Data Cache Lockdown Register format

The L bits for cache ways 3 to 0 are bits [3:0] respectively.

L = 0	Allocation to the cache way is determined by the standard replacement algorithm (reset state).
L = 1	No allocation is performed to this cache way.

A cache lockdown register must only be changed when it is certain that all outstanding accesses that might cause a cache line fill have completed. For this reason, a Data Synchronization Barrier instruction must be executed before the cache lockdown register is changed.

The following procedure for lock down into a data cache way i, with N cache ways, using Format C, ensures that only the target cache way i is locked down.

This is the architecturally defined method for locking data into caches:

1. Ensure that no processor exceptions can occur during the execution of this procedure, by disabling interrupts. If this is not possible, all code and data used by any exception handlers that can be called must be treated as code and data prior to step 2.
2. Ensure that all data used by the following code, apart from the data that is to be locked down, is either:
   - in an uncachable area of memory
   - in an already locked cache way.
3. Ensure that the data to be locked down is in a Cachable area of memory.
4. Ensure that the data to be locked down is not already in the cache, using cache Clean and/or Invalidate instructions as appropriate.

5. Enable allocation to the target cache way by writing to CP15 c9, with the CRm field set to 0, setting L equal to 0 for bit i and L equal to 1 for all other ways.
6. Ensure that the memory cache line is loaded into the cache by using an LDR instruction to load a word from the memory cache line, for each of the cache lines to be locked down in cache way i.
7. Write to CP15 c9, CRm = c0, setting L to 1 for bit i and restore all the other bits to the values they had before this routine was started.

3.4.22 c10, TLB Lockdown Register

The purpose of the TLB Lockdown Register is to control where hardware page table walks place the TLB entry in either:

- the set associative region of the TLB.
- the lockdown region of the TLB, and if in the lockdown region, which entry to write.

The lockdown region of the TLB contains eight entries. See *TLB organization* on page 5-4 for a description of the structure of the TLB.

The TLB Lockdown Register is:

- in CP15 c10
- 32-bit read/write register
- accessible in privileged modes only.

You can access the TLB Lockdown Register by reading or writing CP15 c10 with the Opcode_2 field set to 0:

```
MRC p15, 0, <Rd>, c10, c0, 0; Read TLB Lockdown victim
MCR p15, 0, <Rd>, c10, c0, 0; Write TLB Lockdown victim
```

Figure 3-36 shows the TLB Lockdown Register format.

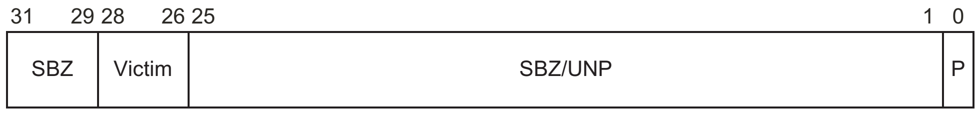


Figure 3-36 TLB Lockdown Register format

Writing the TLB Lockdown Register with the preserve bit (P bit) set to:

1 Means subsequent hardware page table walks place the TLB entry in the lockdown region at the entry specified by the victim, in the range 0 to 7.

0	Means subsequent hardware page table walks place the TLB entry in the set associative region of the TLB.

3.4.23 c10, Memory remap registers

The MMU remap capability has the following form. The remapping is applied to all sources of TLB requests.

The memory region remap registers are accessed by:

```
MCR/MRC{cond} p15, 0, Rd, c10, c2, 0;access primary memory region remap register
```

```
MCR/MRC{cond} p15, 0, Rd, c10, c2, 1;access normal memory region remap register
```

These registers are used to remap memory region types. This remapping is enabled when bit[28] of the CP15 Control Register is set. The remapping takes place on the page table values, and overrides the settings specified in the MMU page tables, or the default behavior when the MMU is turned off. See *Page table descriptors when using remapping* on page 5-18.

The remap capability falls into two levels, the primary remap, allows the primary memory type (Normal, Device or Strongly ordered) to be remapped. For Device and Normal memory, the effect of the S bit can be independently remapped.

After this primary remapping is performed any region that is mapped as Normal memory can have the inner and outer cachable attributes determined by the Normal memory Remap register. To provide maximum flexibility, this level of remapping allows regions that were originally not Normal memory to be remapped independently.

The encoding used for each region is shown in Table 3-35 and *Inner or outer region type encodings* on page 3-62.

Table 3-35 shows the primary remapping encodings.

Table 3-35 Primary remapping encodings

Region	Encoding
Strongly Ordered	00
Shared Device	01
Normal Memory	10
Unpredictable	11

Table 3-36 shows the primary region type encodings.

Table 3-36 Inner or outer region type encodings

Inner or Outer Region	Encoding
noncachable	00
WriteBack, WriteAllocate	01
WriteThrough, Non-WriteAllocate	10
WriteBack, Non-WriteAllocate	11

The primary region remap register determines the memory type and also the treatment of the shareable attribute, and the subsequent normal memory remap register applies to memory regions which, after the primary region remap, are normal memory. This normal memory region remap register allows remapping of the inner cachable and outer cachable attributes.

The fields used for the primary region remap are shown in Table 3-37, and those for the normal memory region remap in Table 3-38 on page 3-63.

Table 3-37 Fields for primary region remap

Bits	Meaning	Reset Value
[31:20]	SBZ/UNP	RAZ
[19]	Remaps shareable attribute when S = 1, for Normal regions	1
[18]	Remaps shareable attribute when S = 0, for Normal regions	0
[17]	Remaps shareable attribute when S = 1, for Device regions	0
[16]	Remaps shareable attribute when S = 0, for Device regions	1
[15:14]	Remaps {TEX[0], C, B} = 111	10
[13:12]	SBZ/UNP	RAZ
[11:10]	Remaps {TEX[0], C, B} = 101	10
[9:8]	Remaps {TEX[0], C, B} = 100	10
[7:6]	Remaps {TEX[0], C, B} = 011	10

Table 3-37 Fields for primary region remap

Bits	Meaning	Reset Value
[5:4]	Remaps {TEX[0], C, B} = 010	10
[3:2]	Remaps {TEX[0], C, B} = 001	01
[1:0]	Remaps {TEX[0], C, B} = 000	00

The reset value for each field is such that no remapping occurs.

Table 3-38 Fields for normal memory region remap

Bits	Meaning	Reset Value
[31:30]	Remaps Outer attribute for {TEX[0], C, B} = 111	01
[29:28]	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED
[27:26]	Remaps Outer attribute for {TEX[0], C, B} = 101	01
[25:24]	Remaps Outer attribute for {TEX[0], C, B} = 100	00
[23:22]	Remaps Outer attribute for {TEX[0], C, B} = 011	11
[21:20]	Remaps Outer attribute for {TEX[0], C, B} = 010	10
[19:18]	Remaps Outer attribute for {TEX[0], C, B} = 001	00
[17:16]	Remaps Outer attribute for {TEX[0], C, B} = 000	01
[15:14]	Remaps Inner attribute for {TEX[0], C, B} = 111	01
[13:12]	-	RAZ
[11:10]	Remaps Inner attribute for {TEX[0], C, B} = 101	10
[9:8]	Remaps Inner attribute for {TEX[0], C, B} = 000	
[7:6]	Remaps Inner attribute for {TEX[0], C, B} = 001	
[5:4]	Remaps Inner attribute for {TEX[0], C, B} = 010	
[3:2]	Remaps Inner attribute for {TEX[0], C, B} = 011	
[1:0]	Remaps Inner attribute for {TEX[0], C, B} = 100	

The reset value for each field is such that no remapping occurs.

The remap registers are expected to be static throughout operation. In particular, when the remap registers are changed, it is Implementation Defined when the changes take effect, it is expected that an invalidation of the TLB and an Instruction Memory Barrier should be performed before any change of the Remap registers can be relied upon.

The Shared bit can also be remapped. If the Shared bit as read from the TLB or page tables is 0, then it is remapped to bit 15 of this register. If the Shared bit as read from the TLB or page tables is 1, then it is remapped to bit 16 of this register.

The reset value for each field ensures that by default no remapping occurs.

Table 3-39 shows the condition of the memory regions, or types, when the MMU is disabled prior to remapping.

Table 3-39 Default memory regions when MMU is disabled

Condition	Region type
Data Cache enabled	Data, Strongly Ordered
Data Cache disabled	Data, Strongly Ordered
Instruction Cache enabled	Instruction, Write-back, write-allocate
Instruction Cache disabled	Instruction, Strongly Ordered

This enables different mappings to be selected with the MMU disabled, that cannot be done using only the I, C, and M bits in CP15 c1.

3.4.24 c13, FCSE PID Register

The use of the FCSE PID Register is deprecated. See *c13, Context ID Register* on page 3-67.

You can access the FCSE PID Register by reading or writing CP15 c13 with the Opcode_2 field set to 0:

```
MRC p15, 0, <Rd>, c13, c0, 0; Read FCSE PID Register
```

```
MCR p15, 0, <Rd>, c13, c0, 0; Write FCSE PID Register
```

Reading from the FCSE PID Register returns the value of the process identifier.

Writing the FCSE PID Register updates the process identifier to the value in bits [31:25]. Bits [24:0] Should Be Zero. Figure 3-37 shows the format of the FCSE PID Register.

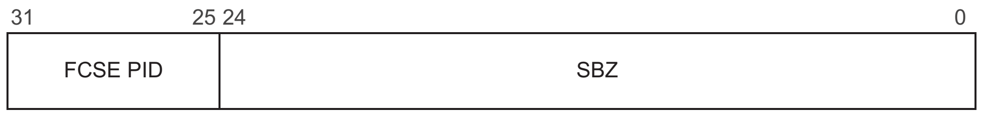


Figure 3-37 FCSE PID Register format

Addresses issued by the ARM11 MPCore processor in the range 0-32MB are translated by the ProcID. Address A becomes A + (ProcID x 32MB). This translated address is used by the MMU. Addresses above 32MB are not translated. The ProcID is a seven-bit field, enabling 64 x 32MB processes to be mapped.

Note

If ProcID is 0, as it is on Reset, then there is a flat mapping between the integer core and the MMU.

Note

Changing the FCSE PID Register value means that the virtual to physical address mapping is changed, so the BTAC must be flushed.

Figure 3-38 on page 3-66 shows how addresses are mapped using CP15 c13.

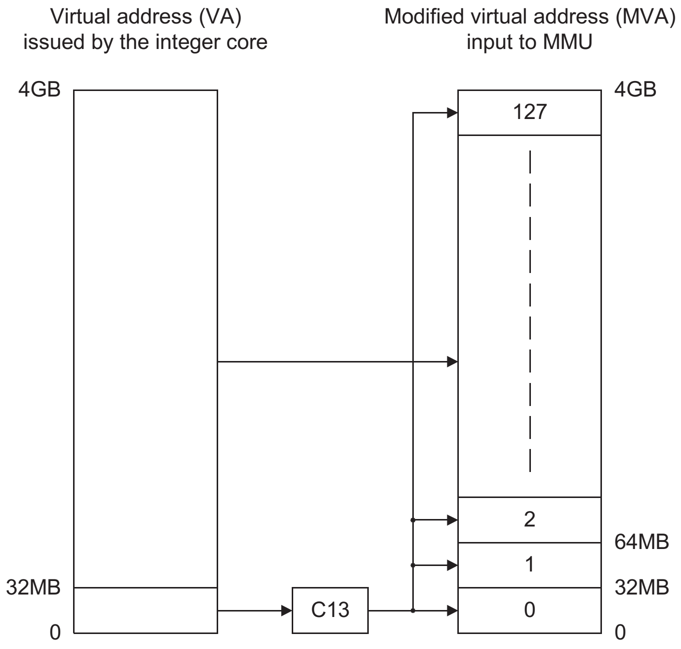


Figure 3-38 Address mapping using CP15 c13

Changing the ProcID, performing a fast context switch

A fast context switch is performed by writing to CP15 c13 FCSE PID Register. The contents of the TLBs do not have to be flushed after a fast context switch because they still hold valid address tags.

From zero to six instructions after the MCR used to write the ProcID might have been fetched with the old ProcID:

```
{ProcID = 0}
MOV r0, #1; Fetched with ProcID = 0
MCR p15,0,r0,c13,c0,0; Fetched with ProcID = 0
A0(any instruction); Fetched with ProcID = 0/1
A1(any instruction); Fetched with ProcID = 0/1
A2(any instruction); Fetched with ProcID = 0/1
A3(any instruction); Fetched with ProcID = 0/1
A4(any instruction); Fetched with ProcID = 0/1
A5(any instruction); Fetched with ProcID = 0/1
A6(any instruction); Fetched with ProcID = 1
```

You must not rely on this behavior for future compatibility. An IMB must be executed between changing the ProcID and fetching from locations that are transmitted by the ProcID.

3.4.25 c13, Context ID Register

The purpose of the Context ID Register is to provide information on the current ASID and process ID, for debug logic, for example. Debug logic uses the ASID information to enable process-dependent breakpoints and watchpoints.

The Context ID Register is:

- in CP15 c13
- a 32-bit read/write register
- accessible in privileged modes only.

You can access the Context ID Register by reading or writing CP15 c13 with the Opcode_2 field set to 1:

```
MRC p15, 0, <Rd>, c13, c0, 1; Read Context ID Register
```

```
MCR p15, 0, <Rd>, c13, c0, 1; Write Context ID Register
```

Figure 3-39 shows the format of the Context ID Register.

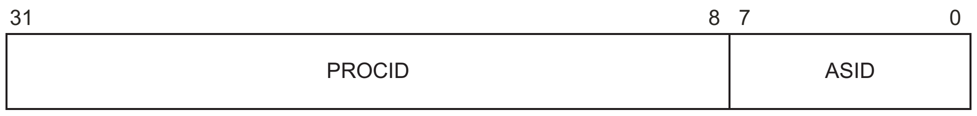


Figure 3-39 Context ID Register format

The bottom eight bits of the Context ID Register are used for the current ASID that is running. The top bits extend the ASID. To ensure that all accesses are related to the correct context ID, you must ensure that software executes a Data Synchronization Barrier operation before changing this register.

The value of this register can also be used to enable process-dependent breakpoints and watchpoints. After changing this register, an IMB sequence must be executed before any instructions are executed that are from an ASID-dependent memory region. Code that updates the ASID must be executed from a global memory region.

Note

Changing the Context ID Register value means that the virtual to physical address mapping is changed, so the BTAC must be flushed.

3.4.26 c13, Thread ID registers

The purpose of the thread and process ID registers is to provide locations to store the IDs of software threads and processes for OS management purposes.

The thread and process ID registers are:

- in CP15 c13
- three 32-bit read/write registers
  - User Read/Write Thread and Process ID Register
  - User Read Only Thread and Process ID Register
  - Privileged Only Thread and Process ID Register.
- each accessible in different modes:
  - User Read/Write: read/write in User and privileged modes
  - User Read Only: read only in User mode, read/write in privileged modes
  - Privileged Only: read/write in privileged modes only.

You can access the thread registers by reading or writing to CP15 c13 with the Opcode_2 field set to 2, 3 or 4:

```
MRC p15,0,<Rd>,c13,c0,2/3/4; Read Thread ID registers
```

```
MCR p15,0,<Rd>,c13,c0,2/3/4; Write Thread ID registers
```

Figure 3-40 shows the Thread ID registers format.

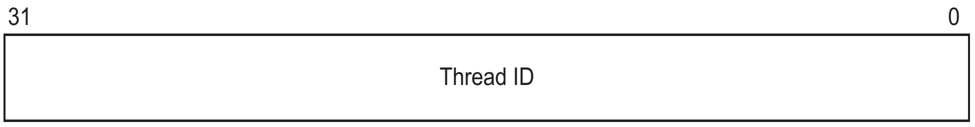


Figure 3-40 Thread ID registers format

Thread ID registers have different access rights depending on Opcode_2 field value:

- Opcode_2 = 2: This register is both user and privileged R/W accessible.
- Opcode_2 = 3: This register is user read-only and privileged R/W accessible.
- Opcode_2 = 4: This register is privileged R/W accessible only.

3.4.27 c15, Performance Monitor Control Register (PMNC)

The Performance Monitor Control Register controls the operation of the Count Register 0 (PMN0), Count Register 1 (PMN1), and Cycle Counter Register (CCNT). This register:

- controls which events PMN0 and PMN1 monitor

- detects which counter overflowed
- enables and disables interrupt reporting
- extends CCNT counting by six more bits (cycles between counter rollover = 2^{38})
- resets all counters to zero
- enables the entire performance monitoring mechanism.

You can access the Performance Monitor Control Register by reading or writing CP15 c15 with the Opcode_2 field set to 0 and the CRm field set to c12:

```
MRC p15, 0, <Rd>, c15, c12, 0; Read Performance Monitor Control Register
```

```
MCR p15, 0, <Rd>, c15, c12, 0 ; Write Performance Monitor Control Register
```

Figure 3-41 shows the format of the Performance Monitor Control Register.

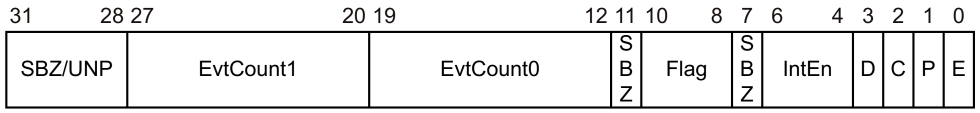


Figure 3-41 Performance Monitor Control Register format

Table 3-40 shows the functions of the bit fields in the Performance Monitor Control Register.

Table 3-40 Performance Monitor Control Register bit functions

Bits	Name	Function
[31:28]	-	SBZ/RAZ
[27:20]	EvtCount1	Identifies the source of events for Count Register 0, as defined in Table 3-41 on page 3-71.
[19:12]	EvtCount0	Identifies the source of events for Count Register 1, as defined in Table 3-41 on page 3-71.
[11]	-	SBZ/RAZ

Table 3-40 Performance Monitor Control Register bit functions (continued)

Bits	Name	Function
[10:8]	Flag	Overflow/Interrupt Flag. Identifies which counter overflowed: Bit 10 = Cycle Counter Register overflow flag Bit 9 = Count Register 1 overflow flag Bit 8 = Count Register 0 overflow flag. For reads: 0 = no overflow (reset) 1 = overflow has occurred. For writes: 0 = no effect 1 = clear this bit.
[6:4]	IntEn	Interrupt Enable. Used to enable and disable interrupt reporting for each counter: Bit 6 = Cycle Counter interrupt enable Bit 5 = Count Register 1 interrupt enable Bit 4 = Count Register 0 interrupt enable. For these registers: 0 = disable interrupt 1 = enable interrupt.
[3]	D	Cycle count divider: 0 = Cycle Counter Register counts every processor clock cycle 1 = Cycle Counter Register counts every 64th processor clock cycle.
[2]	C	Cycle Counter Register Reset on Write, UNP on Read: 0 = no action 1 = reset the Cycle Counter Register to `0x0`.
[1]	P	Count Register Reset on Write, UNP on Read: 0 = no action 1 = reset both Count Registers to `0x0`.
[0]	E	Enable: 0 = all three counters disabled 1 = all three counters enabled.

If an interrupt is generated by this unit, the ARM11 MPCore processor pin **PMUIRQ** is asserted. This output pin can then be routed to an external interrupt controller for prioritization and masking. This is the only mechanism by which the interrupt is signaled to the core.

There is a delay of three cycles between enabling the counter and the counter starting to count events. In addition, the information used to count events is taken from various pipeline stages. This means that the absolute counts recorded might vary because of pipeline effects. This has a negligible effect except in case where the counters are enabled for a very short time.

Table 3-41 shows the events that can be monitored using the Performance Monitor Control Register.

Table 3-41 Performance monitoring events

Event number	Event definition
`0x00`	Instruction cache miss to a cachable location requires fetch from external memory.
`0x01`	Stall because instruction buffer cannot deliver an instruction. This could indicate an Instruction Cache miss or an Instruction MicroTLB miss. This event occurs every cycle in which the condition is present.
`0x02`	Stall because of a data dependency. This event occurs every cycle in which the condition is present.
`0x03`	Instruction MicroTLB miss.
`0x04`	Data MicroTLB miss.
`0x05`	Branch instruction executed, branch might or might not have changed program flow.
`0x06`	Branch not predicted.
`0x07`	Branch mispredicted
`0x08`	Instruction executed.
`0x09`	Folded instruction executed
`0x0A`	Data cache read access, not including Cache operations. This event occurs for each non-sequential access to a cache line.
`0x0B`	Data cache miss, not including Cache Operations.
`0x0C`	Data cache write access.
`0x0D`	Data cache write miss.
`0x0E`	Data cache line eviction, not including cache operations
`0x0F`	Software changed the PC and there is not a mode change.
`0x10`	Main TLB miss.
`0x11`	External memory request. (Cache Refill, Noncachable, Write-Back).

Table 3-41 Performance monitoring events (continued)

Event number	Event definition
`0x12`	Stall because of Load Store Unit request queue being full.
`0x13`	The number of times the Store buffer was drained because of LSU ordering constraints or CP15 operations.
`0x14`	Buffered write merged in a store buffer slot.
`0x15`	LSU is in safe mode.
`0x16`	STB deadlock condition detected
`0xFF`	An increment each cycle.
All other values	Reserved. Unpredictable behavior.

3.4.28 c15, Cycle Counter Register (CCNT)

You can use the Cycle Counter Register to count the core clock cycles. It is a 32-bit counter that can trigger an interrupt on overflow. You can use it in conjunction with the Performance Monitor Control Register and the two Counter Registers to provide a variety of useful metrics that enable you to optimize system performance.

You can access the Cycle Counter Register by reading or writing CP15 c15 with the Opcode_2 field set to 1:

```
MRC p15, 0, <Rd>, c15, c12, 1 ; Read Cycle Counter Register
MCR p15, 0, <Rd>, c15, c12, 1 ; Write Cycle Counter Register
```

The value in the Cycle Counter Register is Unpredictable at Reset. The counter can be set to zero by the Performance Monitor Control Register.

The Cycle Counter Register can be configured to count every 64th clock cycle by the Performance Monitor Control Register.

3.4.29 c15, Count Register 0 (PMN0)

You can use the two counter registers, Count Register 0 and Count Register 1, to count the instances of two different events selected from a list of events by the Performance Monitor Control Register. Each counter is a 32-bit counter that can trigger an interrupt on overflow. By combining different statistics you can obtain a variety of useful metrics that enable you to optimize system performance.

You can access Count Register 0 by reading or writing CP15 c15 with the Opcode_2 field set to 2:

```
MRC p15, 0, <Rd>, c15, c12, 2 ; Read Count Register 0
MCR p15, 0, <Rd>, c15, c12, 2 ; Write Count Register 0
```

The value in Count Register 0 is 0 at Reset.

3.4.30 c15, Count Register 1 (PMN1)

You can use the two counter registers, Count Register 0 and Count Register 1, to count the instances of two different events selected from a list of events by the Performance Monitor Control Register. Each counter is a 32-bit counter that can trigger an interrupt on overflow. By combining different statistics you can obtain a variety of useful metrics that enable you to optimize system performance.

You can access Count Register 1 by reading or writing CP15 c15 with the Opcode_2 field set to 3:

```
MRC p15, 0, <Rd>, c15, c12, 3 ; Read Count Register 1
MCR p15, 0, <Rd>, c15, c12, 3 ; Write Count Register 1
```

The value in Count Register 1 is 0 at Reset.

3.4.31 c15, TLB Debug Control Register

The debug architecture for the ARM11 MPCore processor is described in Chapter 12 *Debug*. The External Debug Interface is based on JTAG, and is described in Chapter 13 *Debug Test Access Port*.

Table 3-42 shows the CP15 c15 operations used for the debug of the main TLB.

Table 3-42 Main TLB debug operations

Function	Data	Instruction
Read TLB Debug Control Register	Data	`MRC p15, 7, <Rd>, c15, c1, 0`
Write to TLB Debug Control Register	Data	`MCR p15, 7, <Rd>, c15, c1, 0`

TLB Debug Control Register

You can disable the loading of the main TLB after a hardware page table walk using the TLB Debug Control Register in CP15 c15.

When the loading of the main TLB is disabled, then misses do not result in the main TLB being updated. This has a significant impact on performance, but enables debug operations to be performed in as unobtrusive a manner as possible.

Figure 3-42 shows the format of the TLB Debug Control Register.

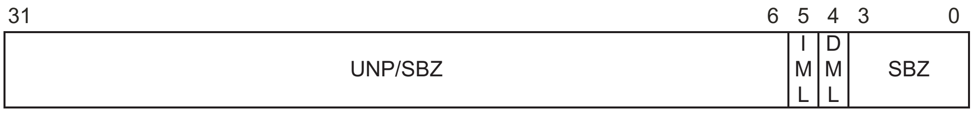


Figure 3-42 TLB Debug Control Register format

Table 3-43 describes the functions of the TLB Debug Control Register bits.

Table 3-43 TLB Debug Control Register bit functions

Bits	Reset value	Name	Description
[31:6]	UNP/SBZ	-	Reserved
[5]	0	IML	1 = Instruction main TLB load disabled 0 = Instruction main TLB load enabled
[4]	0	DML	1 = Data main TLB load disabled 0 = Data main TLB load enabled
[3:0}	UNP/SBZ	-	Reserved

Because the ARM11 MPCore processor has a unified main TLB the IML bit must be set to the same as the DML bit, or else the effect is Unpredictable.

3.4.32 c15, TLB lockdown operations

TLB Lockdown operations allow saving or restoring lockdown entries in the TLB when entering or exiting CPU Dormant mode. Table 3-44 shows the defined TLB lockdown operations.

Table 3-44 TLB lockdown operations

Function	Data	Function
Select Lockdown TLB Entry for Read	Main TLB Index	`MCR p15,5,<Rd>,c15,c4,2`
Select Lockdown TLB Entry for Write	Main TLB Index	`MCR p15,5,<Rd>,c15,c4,4`
Read Lockdown TLB VA Register	Data	`MRC p15,5,<Rd>,c15,c5,2`

Table 3-44 TLB lockdown operations (continued)

Function	Data	Function
Write Lockdown TLB VA Register	Data	`MCR p15,5,<Rd>,c15,c5,2`
Read Lockdown TLB PA Register	Data	`MRC p15,5,<Rd>,c15,c6,2`
Write Lockdown TLB PA Register	Data	`MCR p15,5,<Rd>,c15,c6,2`
Read Lockdown TLB attributes Register	Data	`MRC p15,5,<Rd>,c15,c7,2`
Write Lockdown TLB attributes Register	Data	`MCR p15,5,<Rd>,c15,c7,2`

The Select Lockdown TLB entry for Read operation is used to select from which entry the data read by read Lockdown TLB VA/PA/attributes operations are coming from. The Select Lockdown TLB entry for Write operation is used to select in which entry the data write Lockdown TLB VA/PA/attributes data are written. The TLB PA register must be the last written/read register when accessing TLB lockdown registers. Figure 3-43 shows the format of the index register used to access the lockdown TLB entries.

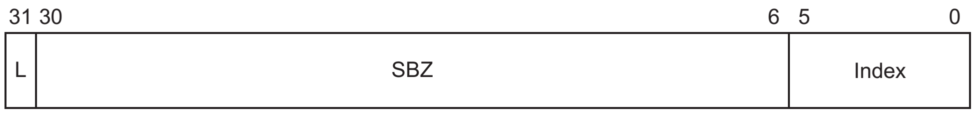


Figure 3-43 Lockdown TLB index format

Figure 3-44 shows the TLB VA Register format.

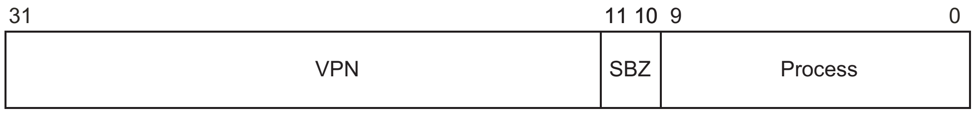


Figure 3-44 TLB VA Register format

Table 3-45 describes the functions of the TLB VA Register bits.

Table 3-45 TLB VA Register bit assignment

Bit	Name	Function
[31:12]	VPN	Virtual page number Bits of the virtual page number that are not translated as part of the page table translation because the size of the tables is Unpredictable when read and SBZ when written.
[11:10]	-	Reserved (RAZ / SBZ)
[9:0]	Process	Memory space identifier that determines if the entry is a global mapping (Process = 0x200), or an ASID dependant entry (Process = {b00, ASID})

Figure 3-45 shows the format of the memory space identifier.

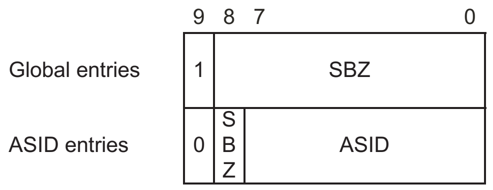


Figure 3-45 Memory space identifier format

Figure 3-46 shows the TLB PA Register format.

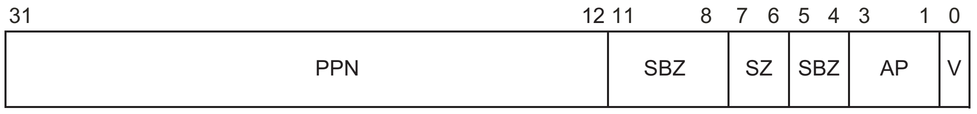


Figure 3-46 TLB PA Register format

Table 3-46 describes the functions of the TLB PA Register bits.

Table 3-46 TLB PA Register bit assignments

Bit	Name	Function
[31:12]	PPN	Physical Page Number. Bits of the physical page number that are not translated as part of the page table translation are unpredictable when read and SBZ when written.
[11:8]	-	Reserved. SBZ / RAZ
[7:6]	SZ	Region Size. b01 : 4KB page b10 : 64KB page b11 : 1MB section b00 : 16MB Supersection All other values are reserved.
[5:4]	-	Reserved. SBZ / RAZ
[3:1]	AP	Access permission: b000 : All access generate a permission fault. b001 : Supervisor access only, User access generates a fault. b010 : Supervisor read/write access, User write access generated a fault b011: Full access, no fault generated b100 : Reserved b101 : Supervisor read only b110 : Supervisor/User read only b111 : Supervisor/User read only
0	V	Value bit. Indicates that this entry is locked and valid.

Figure 3-47 shows the format of the TLB Attributes Register.

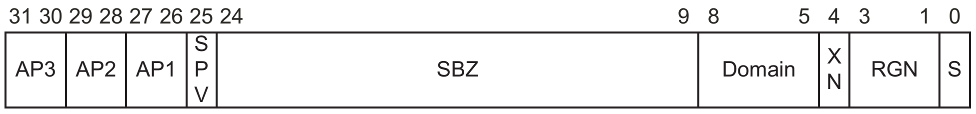


Figure 3-47 TLB Attributes Register

Table 3-47 describe the functions of the TLB Attributes Register bits.

Table 3-47 TLB Attributes Register bit assignments

Bit	Name	Function
[31:30]	AP3	Subpage access permissions for the fourth subpage if the page or section supports subpages. The format for the permissions is shown as the upper subpage permissions in Table 3-48 on page 3-79.
[29:28]	AP2	Subpage access permissions for the third subpage if the page or section supports subpages. The format for the permissions is shown as the upper subpage permissions in Table 3-48 on page 3-79
[37:26]	AP1	Subpage access permissions for the second subpage if the page or section supports subpages. The format for the permissions is shown as the upper subpage permissions in Table 3-48 on page 3-79
[25]	SPV	Subpage valid Indicates that the page or section supports subpages. Pages that support subpages must be marked as Global. 0 = Subpages are not supported. 1 = Subpages are supported.
[24:11]	-	SBZ
[10:7]	Domain	Domain number of the TLB entry.
[6]	XN	Execute Never attribute.
[5:3]	TEX	Region type Encoding. *C and B bit, and type extension field encodings* on page 5-15 for encoding descriptions.
[2:1]	CB	
[0]	S	Shared attribute.

Table 3-48 Upper subpage permissions

Upper subpage permissions AP[1:0]	CP15		Description
	S	R	
b00	0	0	All accesses generate a permission fault
b00	1	0	Supervisor read only. User no access
b00	0	1	Supervisor and User read-only
b00	1	1	Unpredictable

Table 3-48 Upper subpage permissions

Upper subpage permissions AP[1:0]	CP15		Description
	S	R	
b01	X	X	Supervisor access only
b10	X	X	Supervisor full access. User read-only
b11	X	X	Full access

3.5 Summary of CP15 instructions

Table 3-49 shows the CP15 instructions that you can use arranged numerically.

Table 3-49 CP15 instruction summary

Instruction	Operation	Reference
`MRC p15, 0, <Rd>, c0, c0, 0`	Read ID Code Register	page 3-12
`MRC p15, 0, <Rd>, c0, c0, 1`	Read Cache Type Register	page 3-12
`MRC p15, 0, <Rd>, c0, c0, 3`	Read TLB Type Register	page 3-14
`MRC p15, 0, <Rd>, c0, c0, 5`	Read CPU ID Register	page 3-15
`MRC p15, 0, <Rd>, c0, c0, 0`	Read Proc Feature Register 0	page 3-16
`MRC p15, 0, <Rd>, c0, c1, 1`	Read Proc Feature Register 1	page 3-16
`MRC p15, 0, <Rd>, c0, c1, 2`	Read Debug Feature Register 0	page 3-16
`MRC p15, 0, <Rd>, c0, c1, 4`	Read Memory Feature Register 0	page 3-16
`MRC p15, 0, <Rd>, c0, c1, 5`	Read Memory Feature Register 1	page 3-16
`MRC p15, 0, <Rd>, c0, c1, 6`	Read Memory Feature Register 2	page 3-16
`MRC p15, 0, <Rd>, c0, c1, 7`	Read Memory Feature Register 3	page 3-16
`MRC p15, 0, <Rd>, c0, c2, 0`	Read ISA Feature Register 0	page 3-24
`MRC p15, 0, <Rd>, c0, c2, 1`	Read ISA Feature Register 1	page 3-24
`MRC p15, 0, <Rd>, c0, c2, 2`	Read ISA Feature Register 2	page 3-24
`MRC p15, 0, <Rd>, c0, c2, 3`	Read ISA Feature Register 3	page 3-24
`MRC p15, 0, <Rd>, c0, c2, 4`	Read ISA Feature Register 4	page 3-24
`MRC p15, 0, <Rd>, c1, c0, 0`	Read Control Register	page 3-30
`MCR p15, 0, <Rd>, c1, c0, 0`	Write Control Register	page 3-30
`MRC p15, 0, <Rd>, c1, c0, 1`	Read Auxiliary Control Register	page 3-34
`MCR p15, 0, <Rd>, c1, c0, 1`	Write Auxiliary Control Register	page 3-34
`MRC p15, 0, <Rd>, c1, c0, 2`	Read Coprocessor Access Control Register	page 3-36
`MCR p15, 0, <Rd>, c1, c0, 2`	Write Coprocessor Access Control Register	page 3-36
`MRC p15, 0, <Rd>, c2, c0, 0`	Read Translation Table Base Register 0	page 3-37
`MCR p15, 0, <Rd>, c2, c0, 0`	Write Translation Table Base Register 0	page 3-37
`MRC p15, 0, <Rd>, c2, c0, 1`	Read Translation Table Base Register 1	page 3-39
`MCR p15, 0, <Rd>, c2, c0, 1`	Write Translation Table Base Register 1	page 3-39
`MRC p15, 0, <Rd>, c2, c0, 2`	Read Translation Table Base Control Register	page 3-40
`MCR p15, 0, <Rd>, c2, c0, 2`	Write Translation Table Base Control Register	page 3-40
`MRC p15, 0, <Rd>, c3, c0, 0`	Read Domain Access Control Register	page 3-41
`MCR p15, 0, <Rd>, c3, c0, 0`	Write Domain Access Control Register	page 3-41

Table 3-49 CP15 instruction summary (continued)

Instruction	Operation	Reference
`MRC p15, 0, <Rd>, c5, c0, 0`	Read Data Fault Status Register	page 3-42
`MCR p15, 0, <Rd>, c5, c0, 0`	Write Data Fault Status Register	page 3-42
`MRC p15, 0, <Rd>, c5, c0, 1`	Read Instruction Fault Status Register	page 3-44
`MCR p15, 0, <Rd>, c5, c0, 1`	Write Instruction Fault Status Register	page 3-44
`MRC p15, 0, <Rd>, c6, c0, 0`	Read Fault Address Register	page 3-45
`MCR p15, 0, <Rd>, c6, c0, 0`	Write Fault Address Register	page 3-45
`MRC p15, 0, <Rd>, c6, c0, 1`	Read Watchpoint Fault Address Register	page 3-45
`MCR p15, 0, <Rd>, c6, c0, 1`	Write Watchpoint Fault Address Register	page 3-45
`MCR p15, 0, <Rd>, c7, c0, 4`	Wait For Interrupt	page 3-49
`MRC p15, 0, <Rd>, c7, c4, 0`	PA Register	page 3-53
`MCR p15, 0, <Rd>, c7, c5, 0`	Invalidate Entire Instruction Cache Register	page 3-52
`MCR p15, 0, <Rd>, c7, c5, 1`	Invalidate Instruction Cache Line (using MVA) Register	page 3-49
`MCR p15, 0, <Rd>, c7, c5, 2`	Invalidate Instruction Cache Line (using Index) Register	page 3-49
`MCR p15, 0, <Rd>, c7, c5, 4`	Flush Prefetch Buffer Register	page 3-49
`MCR p15, 0, <Rd>, c7, c5, 6`	Flush Entire Branch Target Cache Register	page 3-49
`MCR p15, 0, <Rd>, c7, c5, 7`	Flush Branch Target Cache Entry Register	page 3-49
`MCR p15, 0, <Rd>, c7, c6, 0`	Invalidate Entire Data Cache Register	page 3-49
`MCR p15, 0, <Rd>, c7, c6, 1`	Invalidate Data Cache Line (using MVA) Register	page 3-49
`MCR p15, 0, <Rd>, c7, c6, 2`	Invalidate Data Cache Line (using Index) Register	page 3-49
`MCR p15, 0, <Rd>, c7, c7, 0`	Invalidate Both Caches Register	page 3-49
`MCR p15, 0, <Rn>, c7, c8, 0`	VA to PA with privileged read permission check Register	page 3-49
`MCR p15, 0, <Rn>, c7, c8, 1`	VA to PA with privileged write permission check Register	page 3-52
`MCR p15, 0, <Rn>, c7, c8, 2`	VA to PA with user read permission check Register	page 3-52
`MCR p15, 0, <Rn>, c7, c8, 3`	VA to PA with user write permission check Register	page 3-52
`MCR p15, 0, <Rd>, c7, c10, 0`	Clean Entire Data Cache Register	page 3-52
`MCR p15, 0, <Rd>, c7, c10, 1`	Clean Data Cache Line (using MVA) Register	page 3-52
`MCR p15, 0, <Rd>, c7, c10, 2`	Clean Data Cache Line (using Index) Register	page 3-52
`MCR p15, 0, <Rd>, c7, c10, 4`	Drain Synchronization Barrier Register	page 3-49
`MCR p15, 0, <Rd>, c7, c10, 5`	Data Memory Barrier Register	page 3-49
`MCR p15, 0, <Rd>, c7, c14, 0`	Clean and Invalidate Entire Data Cache Register	page 3-49
`MCR p15, 0, <Rd>, c7, c14, 1`	Clean and Invalidate Data Cache Line (using MVA) Register	page 3-49
`MCR p15, 0, <Rd>, c7, c14, 2`	Clean and Invalidate Data Cache Line (using Index) Register	page 3-49
		page 3-52
		page 3-52
		page 3-52

Table 3-49 CP15 instruction summary (continued)

Instruction	Operation	Reference
`MCR p15,0, <Rd>, c8, C5, 0`	Invalidate Instruction TLB Register	page 3-55
`MCR p15,0, <Rd>, c8, C5, 1`	Invalidate Instruction TLB Single Entry Register	
`MCR p15,0, <Rd>, c8, C5, 2`	Invalidate Instruction TLB Entry on ASID match Register	
`MCR p15,0, <Rd>, c8, C5, 3`	Invalidate Instruction TLB Single Entry on MVA only Register	
`MCR p15,0, <Rd>, c8, C6, 0`	Invalidate Data TLB Register	
`MCR p15,0, <Rd>, c8, C6, 1`	Invalidate Data TLB Single Entry Register	
`MCR p15,0, <Rd>, c8, C6, 2`	Invalidate Data TLB Entry on ASID match Register	
`MCR p15,0, <Rd>, c8, C6, 3`	Invalidate Data TLB Single Entry on MVA only Register	
`MCR p15,0, <Rd>, c8, C7, 0`	Invalidate Unified TLB Register	
`MCR p15,0, <Rd>, c8, C7, 1`	Invalidate Unified TLB Single Entry Register	
`MCR p15,0, <Rd>, c8, C7, 2`	Invalidate Unified TLB Entry on ASID match Register	
`MCR p15,0, <Rd>, c8, C7, 3`	Invalidate Unified TLB Single Entry on MVA only Register	
`MRC p15, 0, <Rd>, c9, c0, 0`	Read Data Cache Lockdown Register	page 3-58
`MCR p15, 0, <Rd>, c9, c0, 0`	Write Data Cache Lockdown Register	page 3-58
`MRC p15, 0, <Rd>, c10, c0, 0`	Read TLB Lockdown Register	page 3-60
`MCR p15, 0, <Rd>, c10, c0, 0`	Write TLB Lockdown Register	
`MRC p15, 0, <Rd>, c10, c2, 0`	Read Primary Remap Register	
`MCR p15, 0, <Rd>, c10, c2, 0`	Write Primary Remap Register	
`MRC p15, 0, <Rd>, c10, c2, 1`	Read Normal Remap Register	
`MCR p15, 0, <Rd>, c10, c2, 1`	Write Normal Remap Register	
`MRC p15, 0, <Rd>, c13, c0, 0`	Read Process ID Register	page 3-64
`MCR p15, 0, <Rd>, c13, c0, 0`	Write Process ID Register	page 3-64
`MRC p15, 0, <Rd>, c13, c0, 1`	Read Context ID Register	page 3-67
`MCR p15, 0, <Rd>, c13, c0, 1`	Write Context ID Register	page 3-67
`MRC p15, 0, <Rd>, c13, c0, 2`	Read Thread ID User and Privileged Read Write Register	
`MCR p15, 0, <Rd>, c13, c0, 2`	Write Thread ID User and Privileged Read Write Register	
`MRC p15, 0, <Rd>, c13, c0, 3`	Read Thread ID User Read only Register	
`MCR p15, 0, <Rd>, c13, c0, 3`	Write Thread ID User Read only Register	
`MRC p15, 0, <Rd>, c13, c0, 4`	Read Thread ID Privileged Read Write only Register	
`MCR p15, 0, <Rd>, c13, c0, 4`	Write Thread ID Privileged Read Write only Register	

Table 3-49 CP15 instruction summary (continued)

Instruction	Operation	Reference
`MRC p15, 0, <Rd>, c15, c12, 0`	Read Performance Monitor Control Register	page 3-68
`MCR p15, 0, <Rd>, c15, c12, 0`	Write Performance Monitor Control Register	page 3-68
`MRC p15, 0, <Rd>, c15, c12, 1`	Read Cycle Counter Register	page 3-72
`MCR p15, 0, <Rd>, c15, c12, 1`	Write Cycle Counter Register	page 3-72
`MRC p15, 0, <Rd>, c15, c12, 2`	Read Count Register 0	page 3-73
`MCR p15, 0, <Rd>, c15, c12, 2`	Write Count Register 0	page 3-73
`MRC p15, 0, <Rd>, c15, c12, 3`	Read Count Register 1	page 3-73
`MCR p15, 0, <Rd>, c15, c12, 3`	Write Count Register 1	page 3-73
`MCR p15, 5, <Rd>, c15, c4, 2`	Read Main TLB Entry Register	page 3-73
`MCR p15, 5, <Rd>, c15, c4, 4`	Write Main TLB Entry Register	
`MRC p15, 5, <Rd>, c15, c5, 2`	Read Main TLB VA Register	
`MCR p15, 5, <Rd>, c15, c5, 2`	Write Main TLB VA Register	
`MRC p15, 5, <Rd>, c15, c6, 2`	Read Main TLB PA Register	
`MCR p15, 5, <Rd>, c15, c6, 2`	Write Main TLB PA Register	
`MRC p15, 5, <Rd>, c15, c7, 2`	Read Main TLB Attribute Register	
`MCR p15, 5, <Rd>, c15, c7, 2`	Write Main TLB Attribute Register	
`MRC p15, 7, <Rd>, c15, c1, 0`	Read TLB Debug Control Register	page 3-73
`MCR p15, 7, <Rd>, c15, c1, 0`	Write TLB Debug Control Register	

Chapter 4
Unaligned and Mixed-Endian Data Access Support

This chapter describes the unaligned and mixed-endianness data access support for MP11 CPUs. It contains the following sections:

4.1 About unaligned and mixed-endian support

MP11 CPUs execute the ARM architecture v6 instructions that support mixed-endian access in hardware, and assist unaligned data accesses. The extensions to ARMv6 that support unaligned and mixed-endian accesses include the following:

- CP15 register c1 has a U bit that enables unaligned support. This bit was specified as zero in previous architectures, and resets to zero for backwards compatibility.
- Architecturally defined unaligned word and halfword access specification for hardware implementation.
- Byte reverse instructions that operate on general-purpose register contents to support signed/unsigned halfword data values.
- Separate instruction and data endianness, with instructions fixed as little-endian format, naturally aligned.
- A PSR endian control flag, the E-bit, cleared on reset and exception entry, that adds a byte-reverse operation to the entire load and store instruction space as data is loaded into and stored back out of the register file. In previous architectures this Program Status Register bit was specified as zero. It is not set in code written to conform to architectures prior to ARMv6.
- ARM and Thumb instructions to set and clear the E-bit explicitly.
- A byte-invariant addressing scheme to support fine-grain big-endian and little-endian shared data structures, to conform to a shared memory standard.

The original ARM architecture was designed as little-endian. This provides a consistent address ordering of bits, bytes, words, cache lines, and pages, and is assumed by the documentation of instruction set encoding and memory and register bit significance. Subsequently, big-endian support was added to enable big-endian byte addressing of memory. A little-endian nomenclature is used for bit-ordering and byte addressing throughout this manual.

Within the ARM11 MPCore processor MP11 CPUs can be set to different endianness.

4.2 Unaligned access support

Instructions must always be aligned as follows:

- ARM 32-bit instructions must be word boundary aligned (Address [1:0] = b00)
- Thumb 16-bit instructions must be halfword boundary aligned (Address [0] = 0).

Unaligned data access support is described in:

- *Word-invariant mode support*
- *ARMv6 extensions*
- *Word-invariant mode and ARMv6 configurations* on page 4-4
- *Word-invariant data access in ARMv6 (U=0)* on page 4-4
- *Support for unaligned data access in ARMv6 (U=1)* on page 4-5
- *ARMv6 unaligned data access restrictions* on page 4-5.

4.2.1 Word-invariant mode support

For ARM architectures prior to ARM architecture v6, data access to non-aligned word and halfword data was treated as aligned from the memory interface perspective. That is, the address is treated as truncated with Address[1:0] treated as zero for word accesses, and Address[0] treated as zero for halfword accesses.

Load single word ARM instructions are also architecturally defined to rotate right the word aligned data transferred by a non word-aligned access, see the *ARM Architecture Reference Manual*.

Alignment fault checking is specified for processors with architecturally compliant *Memory Management Units* (MMUs), under control of CP15 Register c1 A bit, bit 1. When a transfer is not naturally aligned to the size of data transferred a Data Abort is signaled with an Alignment fault status code, see the *ARM Architecture Reference Manual* for more details.

4.2.2 ARMv6 extensions

ARMv6 adds unaligned word and halfword load and store data access support. When enabled, one or more memory accesses are used to generate the required transfer of adjacent bytes transparently, apart from a potentially greater access time where the transaction crosses a word-boundary.

The memory management specification defines a programmable mechanism to enable unaligned access support. This is controlled and programmed using the CP15 register c1 U bit, bit 22.

Non word-aligned for load and store multiple/double, semaphore, synchronization, and coprocessor accesses always signal Data Abort with an Alignment fault status code when the U bit is set.

Strict alignment checking is also supported in ARMv6, under control of the CP15 register c1 A bit (bit 1) and signals a Data Abort with an Alignment fault status code if a 16-bit access is not halfword aligned or a single 32-bit load/store transfer is not word aligned.

ARMv6 alignment fault detection is a mandatory function associated with address generation rather than optionally supported in external memory management hardware.

4.2.3 Word-invariant mode and ARMv6 configurations

The unaligned access handling is summarized in Table 4-1.

Table 4-1 Unaligned access handling

CP15 register c1 U bit	CP15 register c1 A bit	Unaligned access model
0	0	Word-invariant ARMv5. See *Word-invariant data access in ARMv6 (U=0)*.
0	1	Word-invariant natural alignment check.
1	0	ARMv6 unaligned half/word access, else strict word alignment check.
1	1	ARMv6 strict half/word alignment check.

For a fuller description of the options available, see *c1, Control Register* on page 3-29.

4.2.4 Word-invariant data access in ARMv6 (U=0)

MP11 CPUs emulates earlier architecture unaligned accesses to memory as follows:

- If A bit is asserted alignment faults occur for:

Halfword access	Address[0] is 1.
Word access	Address[1:0] is not b00.
LDRD or STRD	Address [2:0] is not b000.
Multiple access	Address [1:0] is not b00.

- If alignment faults are enabled and the access is not aligned then the Data Abort vector is entered with an Alignment fault status code.

- If no alignment fault is enabled, that is, if bit 1 of CP15 register c1, the A bit, is not set:

Byte access	Memory interface uses full Address [31:0].
Halfword access	Memory interface uses Address [31:1]. Address [0] asserted as 0.
Word access	Memory interface uses Address [31:2]. Address [1:0] asserted as 0.

  — ARM load data rotates the aligned read data and rotates this right by the byte-offset denoted by Address [1:0], see the *ARM Architecture Reference Manual*.

  — ARM and Thumb load-multiple accesses always treated as aligned. No rotation of read data.

  — ARM and Thumb store word and store multiple treated as aligned. No rotation of write data.

  — ARM load and store doubleword operations treated as 64-bit aligned.

  — Thumb load word data operations are Unpredictable if not word aligned.

  — ARM and Thumb halfword data accesses are Unpredictable if not halfword aligned.

4.2.5 Support for unaligned data access in ARMv6 (U=1)

The MP11 CPU memory interfaces can generate unaligned low order byte address offsets only for halfword and single word load and store operations, and byte accesses unless the A bit is set. These accesses produce an alignment fault if the A bit is set, and for some of the cases described in *ARMv6 unaligned data access restrictions*.

If alignment faults are enabled and the access is not aligned then the Data Abort vector is entered with an Alignment Fault status code.

4.2.6 ARMv6 unaligned data access restrictions

The following restrictions apply for ARMv6 unaligned data access:

- Accesses are not guaranteed atomic. They might be synthesized out of a series of aligned operations in a shared memory system without guaranteeing locked transaction cycles.
- Unaligned accesses loading the PC produce an alignment trap.

- Accesses typically take a number of cycles to complete compared to a naturally aligned transfer. The real-time implications must be carefully analyzed and key data structures might require to have their alignment adjusted for optimum performance.
- Accesses can abort on either or both halves of an access where this occurs over a page boundary. The Data Abort handler must handle restartable aborts carefully after an Alignment fault status code is signaled.

As a result, shared memory schemes must not rely on seeing monotonic updates of non-aligned data of loads, stores, and swaps for data items greater than byte width.

Unaligned access operations must not be used for accessing Device memory-mapped registers, and must be used with care in Shared memory structures that are protected by aligned semaphores or synchronization variables.

An Alignment fault occurs if unaligned accesses to Strongly Ordered or Device memory are attempted.

Swap and synchronization primitives, multiple-word or coprocessor access produce an alignment fault regardless of the setting of the A bit.

4.3 Unaligned data access specification

The architectural specification of unaligned data representations is defined in terms of bytes transferred between memory and register, regardless of bus width and bus endianness.

Little-endian data items are described using lower-case byte labeling bX..b0 (byteX to byte 0) and a pointer is always treated as pointing to the least significant byte of the addressed data.

Big-endian data items are described using upper-case byte labeling B0..BX (BYTE0 to BYTEX) and a pointer is always treated as pointing to the most significant byte of the addressed data.

4.3.1 Load unsigned byte, endian independent

The addressed byte is loaded from memory into the low eight bits of the general-purpose register and the upper 24 bits are zeroed. Figure 4-1 shows this.

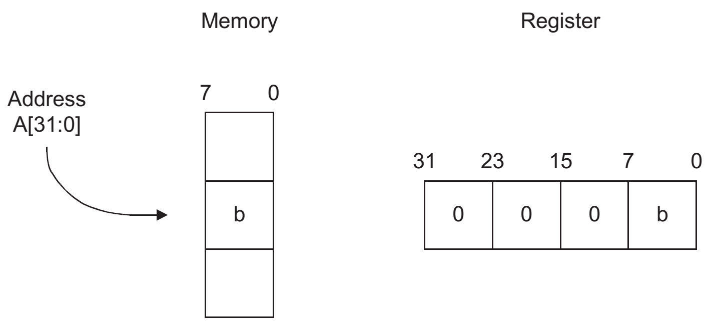


Figure 4-1 Load unsigned byte

4.3.2 Load signed byte, endian independent

The addressed byte is loaded from the memory into the low eight bits of the general-purpose register and the sign bit is extended into the upper 24 bits of the register. Figure 4-2 on page 4-8 shows this.

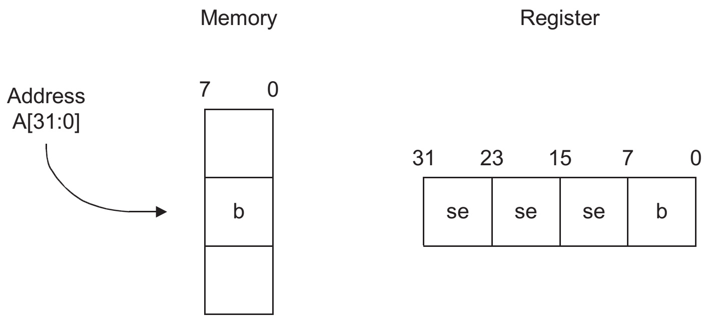


Figure 4-2 Load signed byte

In Figure 4-2, se means b (bit 7) sign extension.

4.3.3 Store byte, endian independent

The low eight bits of the general-purpose register are stored into the addressed byte in memory. Figure 4-3 shows this.

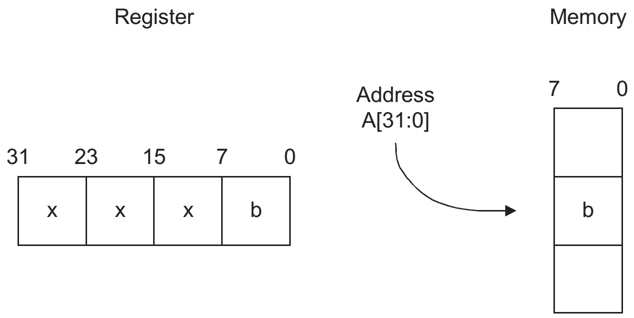


Figure 4-3 Store byte

4.3.4 Load unsigned halfword, little-endian

The addressed byte-pair is loaded from memory into the low 16 bits of the general-purpose register, and the upper 16 bits are zeroed so that the least-significant addressed byte in memory appears in bits [7:0] of the ARM register. Figure 4-4 on page 4-9 shows this.

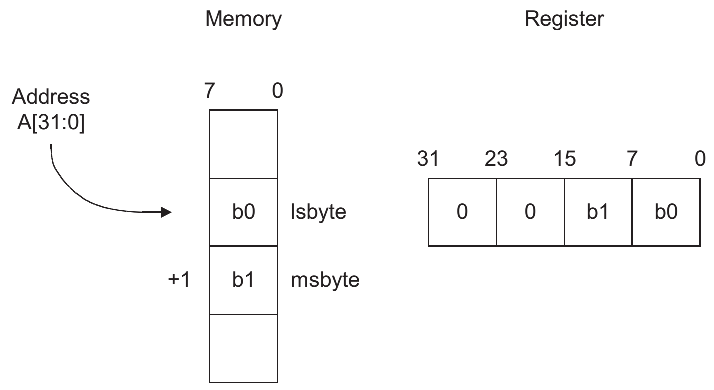


Figure 4-4 Load unsigned halfword, little-endian

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.5 Load unsigned halfword, big-endian

The addressed byte-pair is loaded from memory into the low 16 bits of the general-purpose register, and the upper 16 bits are zeroed so that the most-significant addressed byte in memory appears in bits [15:8] of the ARM register. Figure 4-5 shows this.

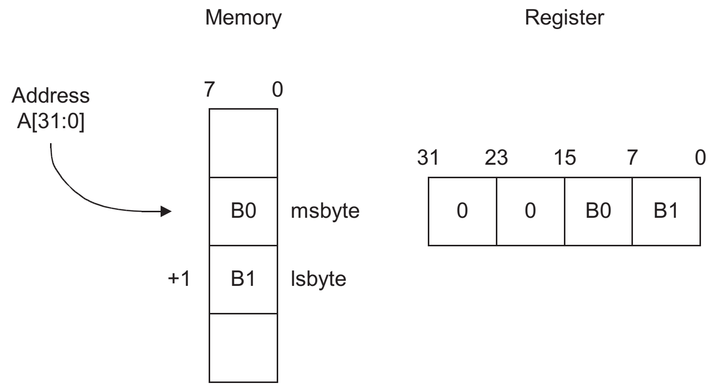


Figure 4-5 Load unsigned halfword, big-endian

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.6 Load signed halfword, little-endian

The addressed byte-pair is loaded from memory into the low 16-bits of the general-purpose register, so that the least-significant addressed byte in memory appears in bits [7:0] of the ARM register and the upper 16 bits are sign-extended from bit 15. Figure 4-6 shows this.

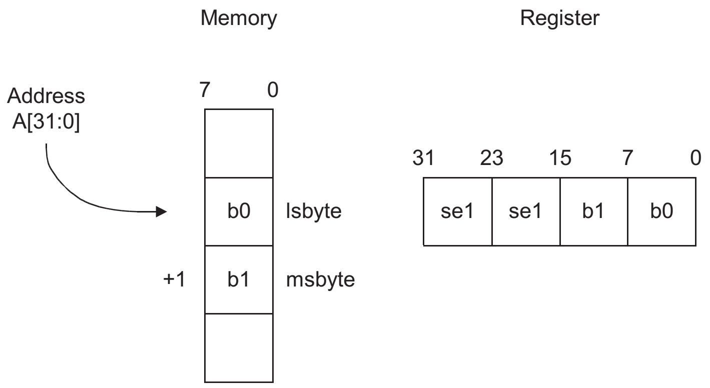


Figure 4-6 Load signed halfword, little-endian

In Figure 4-6, se1 means bit 15 (b1 bit 7) sign extended.

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.7 Load signed halfword, big-endian

The addressed byte-pair is loaded from memory into the low 16-bits of the general-purpose register, so that the most significant addressed byte in memory appears in bits [15:8] of the ARM register and bits [31:16] replicate the sign bit in bit 15. Figure 4-7 on page 4-11 shows this.

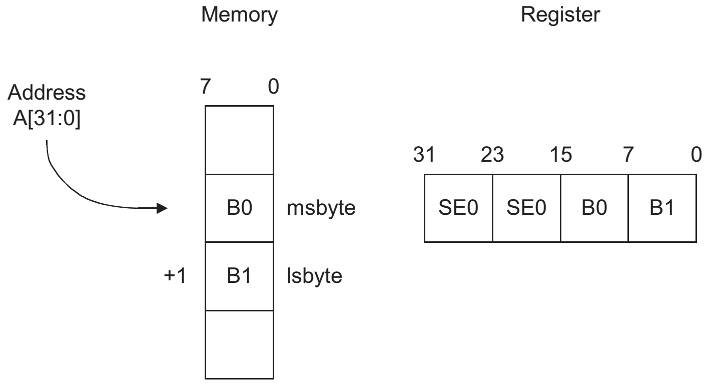


Figure 4-7 Load signed halfword, big-endian

In Figure 4-7, SE0 means bit 15 (B0 bit 7) sign extended.

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.8 Store halfword, little-endian

The low 16 bits of the general-purpose register are stored into the memory with bits [7:0] written to the addressed byte in memory, bits [15:8] to the incremental byte address in memory. Figure 4-8 shows this.

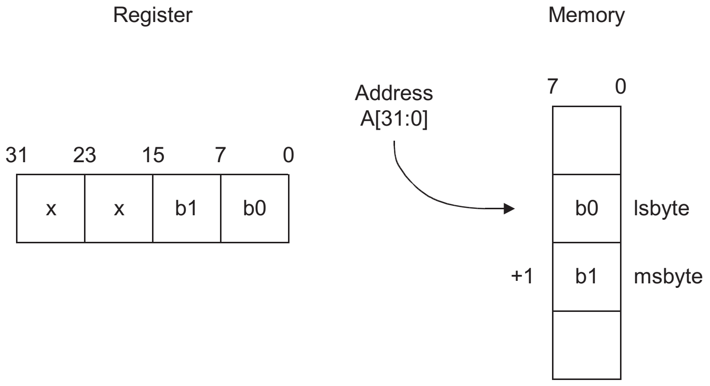


Figure 4-8 Store halfword, little-endian

If strict alignment fault checking is enabled and Address bit 0 is not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.9 Store halfword, big-endian

The low 16 bits of the general-purpose register are stored into the memory with bits [15:8] written to the addressed byte in memory, bits [7:0] to the incremental byte address in memory. Figure 4-9 shows this.

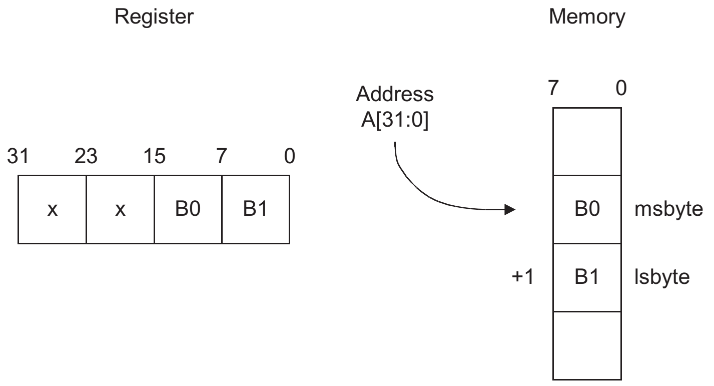


Figure 4-9 Store halfword, big-endian

If strict alignment fault checking is enabled and Address bits [1:0] is not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.10 Load word, little-endian

The addressed byte-quad is loaded from memory into the 32-bit general-purpose register so that the least-significant addressed byte in memory appears in bits [7:0] of the ARM register. Figure 4-10 on page 4-13 shows this.

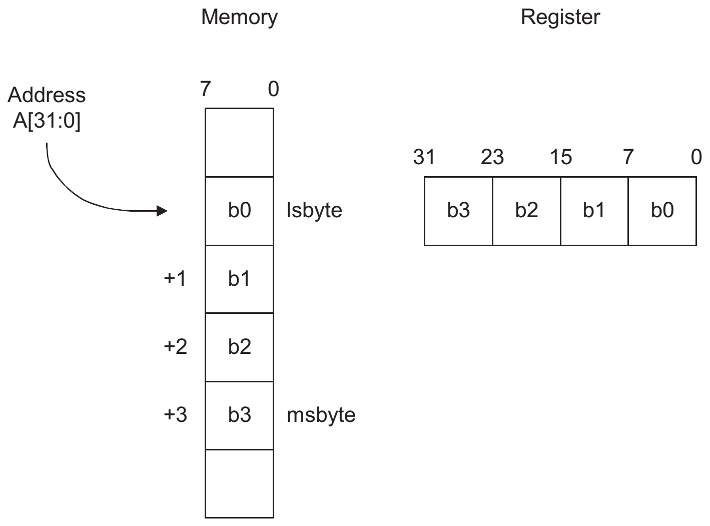


Figure 4-10 Load word, little-endian

If strict alignment fault checking is enabled and Address bits [1:0] is not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.11 Load word, big-endian

The addressed byte-quad is loaded from memory into the 32-bit general-purpose register so that the most significant addressed byte in memory appears in bits [31:24] of the ARM register. Figure 4-11 on page 4-14 shows this.

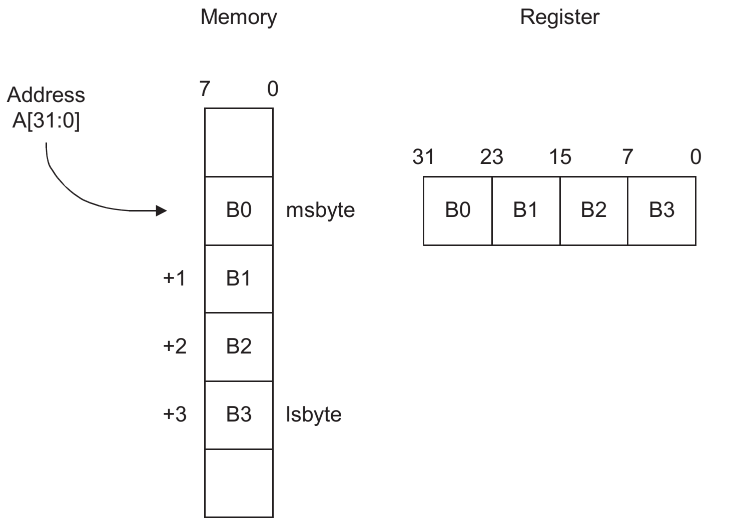


Figure 4-11 Load word, big-endian

If strict alignment fault checking is enabled and Address bits [1:0] is not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.12 Store word, little-endian

The 32-bit general-purpose register is stored to four bytes in memory where bits [7:0] of the ARM register are transferred to the least-significant addressed byte in memory. Figure 4-12 on page 4-15 shows this.

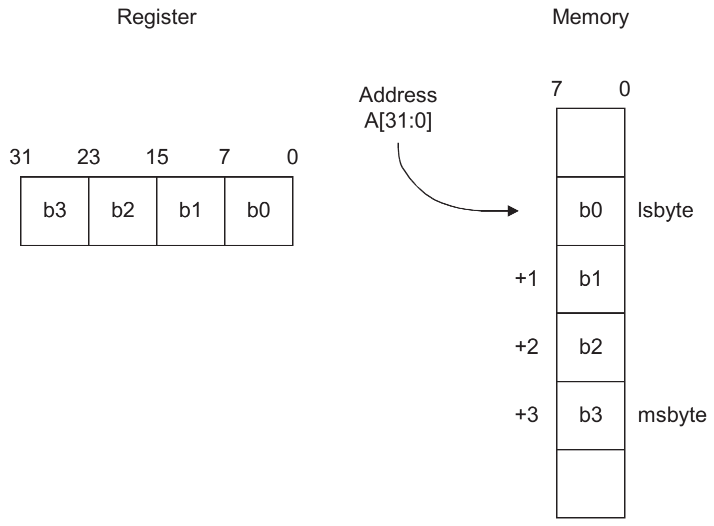


Figure 4-12 Store word, little-endian

If strict alignment fault checking is enabled and Address bits [1:0] are not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.13 Store word, big-endian

The 32-bit general-purpose register is stored to four bytes in memory where bits [31:24] of the ARM register are transferred to the most-significant addressed byte in memory. Figure 4-13 on page 4-16 show this.

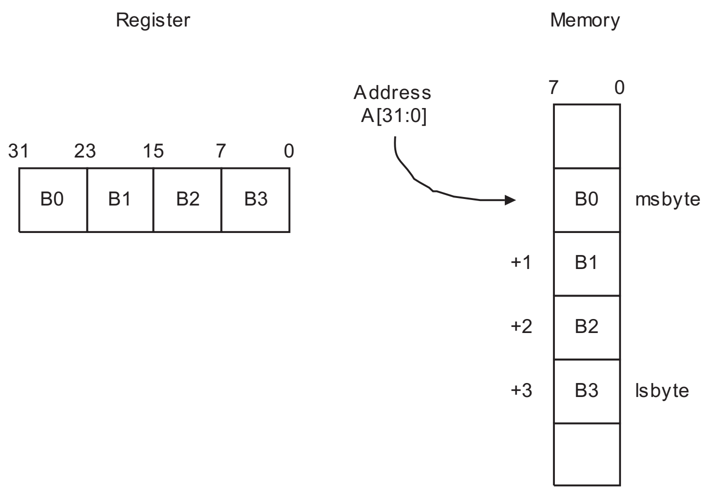


Figure 4-13 Store word, big-endian

If strict alignment fault checking is enabled and Address bits [1:0] are not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.14 Load double, load multiple, load coprocessor (little-endian, E = 0)

The access is treated as a series of incrementing aligned word loads from memory. The data is treated as load word data (see *Load word, little-endian* on page 4-13) where the lowest two address bits are zeroed.

If strict alignment fault checking is enabled and effective Address bits[1:0] are not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.15 Load double, load multiple, load coprocessor (big-endian, E=1)

The access is treated as a series of incrementing aligned word loads from memory. The data is treated as load word data (see *Load word, big-endian* on page 4-14) where the lowest two address bits are zeroed.

If strict alignment fault checking is enabled and effective Address bits[1:0] are not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.16 Store double, store multiple, store coprocessor (little-endian, E=0)

The access is treated as a series of incrementing aligned word stores to memory. The data is treated as store word data (see *Store word, little-endian* on page 4-15) where the lowest two address bits are zeroed.

If strict alignment fault checking is enabled and effective Address bits[1:0] are not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.3.17 Store double, store multiple, store coprocessor (big-endian, E=1)

The access is treated as a series of incrementing aligned word stores to memory. The data is treated as store word data (see *Store word, big-endian* on page 4-16) where the lowest two address bits are zeroed.

If strict alignment fault checking is enabled and effective Address bits[1:0] are not zero, then a Data Abort is generated and the MMU returns an Alignment fault in the Fault Status Register.

4.4 Operation of unaligned accesses

Alignment faults and the operation of non-faulting accesses of the MP11 CPUs are described in this section.

Table 4-3 on page 4-19 gives details of when an Alignment fault must occur for an access and of when the behavior of an access is architecturally Unpredictable. When an access neither generates an Alignment fault and is not Unpredictable, details of precisely which memory locations are accessed are also given in the table.

The access type descriptions used in the Table 4-3 on page 4-19 are determined from the load/store instructions given in Table 4-2.

Table 4-2 Memory access type descriptions

Access type	ARM instructions	Thumb instructions
Byte	LDRB, LDRBT, LDRSB, STRB, STRBT	LDRB, LDRSB, STRB
BSync	SWPB, LDREXB, STREXB	-
Halfword	LDRH, LDRSH, STRH	LDRH, LDRSH, STRH
HWSync	LDREXH, STREXH	-
WLoad	LDR, LDRT, SWP (load access, if U is set to 0)	LDR
WStore	STR, STRT, SWP (store access, if U is set to 0)	STR
WSync	LDREX, STREX, SWP (either access, if U is set to 1)	-
Two-word	LDRD, STRD	-
Multi-word	LDC, LDM, RFE, SRS, STC, STM	LDMIA, POP, PUSH, STMIA
DWSync	LDREXD, STREXD	-

The following terminology is used to describe the memory locations accessed:

Byte[X] This means the byte whose address is X in the current endianness model. The correspondence between the endianness models is that Byte[A] in the LE endianness model, Byte[A] in the BE-8 endianness model.

Halfword[X] This means the halfword consisting of the bytes whose addresses are X and X+1 in the current endianness model, combined to form a halfword in little-endian order in the LE endianness model or in big-endian order in the BE-8 endianness model.

Word[X] This means the word consisting of the bytes whose addresses are X, X+1, X+2, and X+3 in the current endianness model, combined to form a word in little-endian order in the LE endianness model or in big-endian order in the BE-8 endianness model.

Align(X) This means X AND `0xFFFFFFFC`. That is, X with its least significant two bits forced to zero to make it word-aligned.

There is no difference between Addr and Align(Addr) on lines where Addr[1:0] is set to b00. You can use this to simplify the control of when the least significant bits are forced to zero.

For the Two-word and Multi-word access types, the memory accessed column only specifies the lowest word accessed. Subsequent words have addresses constructed by successively incrementing the address of the lowest word by four, and are constructed using the same endianness model as the lowest word.

Table 4-3 Unalignment fault occurrence when access behavior is architecturally unpredictable

A	U	Addr[2:0]	Access types	Architectural Behavior	Memory accessed	Note
0	0	-	-	-	-	Legacy, no alignment
0	0	bxxx	Byte, BSync	Normal	Byte[Addr]	
0	0	bxx0	Halfword	Normal	Halfword[Addr]	
0	0	bxx1	Halfword	Unpredictable	-	
0	0	bxx0	HWSync	Normal	Halfword[Addr]	
0	0	bxx1	HWSync	Unpredictable	-	
0	0	bxxx	Wload	Normal	Word[Align32(Addr)]	Loaded data rotated by 8*Addr[1:0] bits
0	0	bxxx	WStore	Normal	Word[Align32(Addr)]	Operation unaffected by Addr[1:0]
0	0	bx00	WSync	Normal	Word[Addr]	
0	0	bxx1, bx1x	WSync	Unpredictable	-	
0	0	bxxx	Multi-word	Normal	Word[Align32(Addr)]	Operation unaffected by Addr[1:0]
0	0	b000	Two-word	Normal	Word[Addr]	

Table 4-3 Unalignment fault occurrence when access behavior is architecturally unpredictable (continued)

A	U	Addr[2:0]	Access types	Architectural Behavior	Memory accessed	Note
0	0	bxx1, bx1x, b1xx	Two-word	Unpredictable	-	
0	0	b000	DWSync	Normal	Word[Addr]	
0	0	bxx1, bx1x, b1xx	DWSync	Unpredictable	-	
0	1	-	-	-	-	ARMv6 unaligned support[a]
0	1	bxxx	Byte, BSync	Normal	Byte[Addr]	
0	1	bxxx	Halfword	Normal	Halfword[Addr]	
0	1	bxx0	HWSync	Normal	Halfword[Addr]	
0	1	bxx1	HWSync	Alignment fault		
0	1	bxxx	Wload, WStore	Normal	Word[Addr]	
0	1	bx00	WSync, Multi-word, Two-word	Normal	Word[Addr]	
0	1	bxx1, bx1x	WSync, Multi-word, Two-word	Alignment fault	-	-
0	1	b000	DWSync	Normal	Word[Addr]	
0	1	bxx1, bx1x, b1xx	DWSync	Alignment fault	-	
1	x	-	-	-	-	Full alignment faulting
1	x	bxxx	Byte, BSync	Normal	Byte[Addr]	
1	x	bxx0	Halfword, HWSync	Normal	Halfword[Addr]	
1	x	bxx1	Halfword, HWSync	Alignment fault	-	

Table 4-3 Unalignment fault occurrence when access behavior is architecturally unpredictable (continued)

A	U	Addr[2:0]	Access types	Architectural Behavior	Memory accessed	Note
1	x	bx00	WLoad, WStore, WSync, Multi-word	Normal	Word[Addr]	
1	x	bxx1, bx1x	WLoad, WStore, WSync, Multi-word	Alignment fault	-	
1	x	b000	Two-word	Normal	Word[Addr]	
1	0	b100	Two-word	Alignment fault	-	
1	1	b100	Two-word	Normal	Word[Addr]	
1	x	bxx1, bx1x	Two-word	Alignment fault	-	
1	x	b000	DWSync	Normal	Word[Addr]	
1	x	bxx1, bx1x, b1xx	DWSync	Alignment fault	-	

a. Alignment faults occur when unaligned accesses to Strongly Ordered or Device memory are attempted.

The following causes override the behavior specified in the Table 4-3 on page 4-19:

- An LDR instruction that loads the PC, has Addr[1:0] != b00, and is specified in the table as having Normal behavior instead has Unpredictable behavior.

  The reason why this applies only to LDR is that most other load instructions are Unpredictable regardless of alignment if the PC is specified as their destination register.

  The exceptions are the ARM LDM and RFE instructions, and the Thumb POP instruction. If the instruction for them is Addr[1:0] != b00, the effective address of the transfer has its two least significant bits forced to 0 if A is set 0 and U is set to 0. Otherwise the behavior specified in *Unalignment fault occurrence when access behavior is architecturally unpredictable* on page 4-19 is either Unpredictable or an Alignment fault regardless of the destination register.

4.5 Mixed-endian access support

Mixed-endian data access is described in:

- *ARMv6 support for mixed-endian data*
- *Instructions to change the CPSR E bit* on page 4-26.

4.5.1 ARMv6 support for mixed-endian data

Prior to ARMv6 the endianness of both instructions and data are locked together, and the configuration of the processor and the external memory system must either be hard-wired or programmed in the first few instructions of the bootstrap code. In ARMv6 the instruction and data endianness are separated:

- instructions are fixed little-endian
- data accesses can be either little-endian or big-endian as controlled by bit 9, the E bit, of the Program Status Register.

The value of the E bit on any exception entry, including reset, is determined by the CP15 Control Register EE bit.

Fixed little-endian Instructions

Instructions must be naturally aligned and are always treated as being stored in memory in little-endian format. That is, the PC points to the least-significant-byte of the instruction.

Instructions have to be treated as data by exception handlers (decoding SWI calls and Undefined instructions, for example).

Instructions can also be written as data by debuggers, Just-In-Time compilers, or in operating systems that update exception vectors.

Mixed-endian data access

The operating-system typically has a required endian representation of internal data structures, but applications and device drivers have to work with data shared with other processors (DSP or DMA interfaces) that might have fixed big-endian or little-endian data formatting.

A byte-invariant addressing mechanism is provided that enables the load/store architecture to be qualified by the CPSR E bit that provides byte reversing of big-endian data in to, and out of, the processor register bank transparently. This byte-invariant big-endian representation is called BE-8 in this document.

The effect on byte, halfword, word, and multi-word accesses of setting the CPSR E bit when the U bit enables unaligned support is described in *Mixed-endian configuration support* on page 4-24.

Byte data access

The same physical byte in memory is accessed whether big-endian or little-endian:

- Unsigned byte load as described in *Load unsigned byte, endian independent* on page 4-7.
- Signed byte load as described in *Load signed byte, endian independent* on page 4-7.
- Byte store as described in *Store byte, endian independent* on page 4-8.

Halfword data access

The same two physical bytes in memory are accessed whether big-endian or little-endian. Big-endian halfword load data is byte-reversed as read into the processor register to ensure little-endian internal representation, and similarly is byte-reversed on store to memory:

- Unsigned halfword load as described in *Load unsigned halfword, little-endian* on page 4-8 (LE), and *Load unsigned halfword, big-endian* on page 4-9 (BE-8).
- Signed halfword load as described in *Load signed halfword, little-endian* on page 4-10 (LE), and *Load signed halfword, big-endian* on page 4-10 (BE-8).
- Halfword store as described in *Store halfword, little-endian* on page 4-11 (LE), and *Store halfword, big-endian* on page 4-12 (BE-8).

Load Word

The same four physical bytes in memory are accessed whether big-endian or little-endian. Big-endian word load data is byte reversed as read into the processor register to ensure little-endian internal representation, and similarly is byte-reversed on store to memory:

- Word load as described in *Load word, little-endian* on page 4-12 (LE), and *Load word, big-endian* on page 4-13 (BE-8).
- Word store as described in *Store word, little-endian* on page 4-14 (LE), and *Store word, big-endian* on page 4-15 (BE-8).

Mixed-endian configuration support

This behavior is enabled when the U bit in CP15 Register c1 is set. Table 4-4 shows the mixed-endian configurations.

Table 4-4 Mixed-endian configuration

U	E	Instruction endianness	Data endianness	Description
1	0	LE	LE	LE instructions, LE data load/store.
1	1	LE	BE-8	LE instructions, BE data load/store

4.5.2 Reset values of the EE, U, and E bits

The reset values of the EE,U, and E and bits are determined by **CFGEND[1:0]**. Table 4-5 shows this.

Table 4-5 EE bit, U bit, and E bit settings

CFGEND[1:0]	CP15 control register		CPSR and SPSR
	EE	U	E
00	0	0	0
01 Reserved	-	-	-
10	0	1	0
11	1	1	1

4.6 Instructions to reverse bytes in a general-purpose register

When an application or device driver has to interface to memory-mapped peripheral registers or shared-memory DMA structures that are not the same endianness as that of the internal data structures, or the endianness of the Operating System, an efficient way of being able to explicitly transform the endianness of the data is required.

The following new instructions are added to the ARM and Thumb instruction sets to provide this functionality:

- reverse word (4 bytes) register, for transforming big and little-endian 32-bit representations
- reverse halfword and sign-extend, for transforming signed 16-bit representations
- Reverse packed halfwords in a register for transforming big- and little-endian 16-bit representations.

4.6.1 All load and store operations

All load and store instructions take account of the CPSR E bit. Data is transferred directly to registers when E = 0, and byte reversed if E = 1 for halfword, word, or multiple word transfers.

Operation:

```
When CPSR[<E-bit>] = 1 then byte reverse load/store data
```

4.7 Instructions to change the CPSR E bit

ARM and Thumb instructions are provided to set and clear the E-bit efficiently:

`SETEND BE`	Sets the CPSR E bit
`SETEND LE`	Resets the CPSR E bit.

These are specified as unconditional operations to minimize pipelined implementation complexity.

Chapter 5
Memory Management Unit

This chapter describes the *Memory Management Unit* (*MMU*) and how it is used. It contains the following sections:

5.1 About the MMU

Each MP11 CPU MMU works with the cache memory system to control accesses to and from external memory. The MMU also controls the translation of Virtual Addresses to physical addresses.

The ARM11 MPCore processor implements an ARMv6 MMU to provide address translation and access permission checks for the instruction and data ports of the MP11 CPUs. The MMU controls table-walking hardware that accesses translation tables in main memory. A single set of two-level page tables stored in main memory controls the contents of the instruction and data side *Translation Lookaside Buffers* (TLBs). The finished Virtual Address to physical address translation is put into the TLB. The TLBs are enabled from a single bit in CP15 Control Register c1, providing a single address translation and protection scheme from software.

The MMU features are:

- standard ARMv6 MMU mapping sizes, domains, and access protection scheme
- mapping sizes are 4KB, 64KB, 1MB, and 16MB
- the access permissions for 1MB sections and 16MB supersections are specified for the entire section
- you can specify access permissions for 64KB large pages and 4KB small pages separately for each quarter of the page (these quarters are called subpages)
- 16 domains
- one 64-entry unified TLB and a lockdown region of eight entries
- you can mark entries as a global mapping, or associated with a specific application space identifier to eliminate the requirement for TLB flushes on most context switches
- access permissions extended to enable supervisor read-only and supervisor/user read-only modes to be simultaneously supported
- memory region attributes to mark pages shared by multiple processors
- hardware page table walks
- Round-Robin replacement algorithm.

The MMU memory system architecture enables fine-grained control of a memory system. This is controlled by a set of virtual to physical address mappings and associated memory properties held within one or more structures known as TLBs within the MMU. The contents of the TLBs are managed through hardware translation lookups from a set of translation tables in memory.

To prevent requiring a TLB invalidation on a context switch, you can mark each virtual to physical address mapping as being associated with a particular application space, or as global for all application spaces. Only global mappings and those for the current application space are enabled at any time. By changing the *Application Space IDentifier* (ASID) you can alter the enabled set of virtual to physical address mappings. The set of memory properties associated with each TLB entry include:

Memory access permission control

This controls if a program has no-access, read-only access, or read/write access to the memory area. When an access is attempted without the required permission, a memory abort is signaled to the processor. The level of access possible can also be affected by whether the program is running in User mode, or a privileged mode, and by the use of domains. See *Memory access control* on page 5-11 for more details.

Memory region attributes

These describe properties of a memory region. Examples include Device, Noncachable, Write-Through, and Write-Back. If an entry for a Virtual Address is not found in a TLB then a set of translation tables in memory are automatically searched by hardware to create a TLB entry. This process is known as a translation table walk. If the ARM11 MPCore processor is in ARMv5 backwards-compatible mode some new features, such as ASIDs, are not available. The MMU architecture also enables specific TLB entries to be locked down in a TLB. This ensures that accesses to the associated memory areas never require looking up by a translation table walk. This minimizes the worst-case access time to code and data for real-time routines.

5.2 TLB organization

The TLB organization is described in:

- *MicroTLB*
- *Main TLB*
- *TLB control operations* on page 5-5
- *Page-based attributes* on page 5-5
- *Supersections* on page 5-6.

5.2.1 MicroTLB

The first level of caching for the page table information is a small MicroTLB of eight entries that is implemented on each of the instruction and data sides. These entities are implemented in logic, providing a fully associative lookup of the Virtual Addresses in a cycle. This means that a MicroTLB miss signal is returned at the end of the DC1 cycle. In addition to the Virtual Address, an *Address Space IDentifier* (ASID) is used to distinguish different address mappings that might be in use.

The current ASID is a small identifier, eight bits in size, that is programmed using CP15 when different address mappings are required. A memory mapping for a page or section can be marked as being global or referring to a specific ASID. The MicroTLB uses the current ASID in the comparisons of the lookup for all pages for which the global bit is not set.

The MicroTLB returns the physical address to the cache for the address comparison, and also checks the protection attributes in sufficient time to signal a Data Abort in the DC2 cycle. A additional set of attributes, to be used by the cache line miss handler, are provided by the MicroTLB. The timing requirements for these are less critical than for the physical address and the abort checking.

All Main TLB maintenance operations affect both the instruction and data MicroTLBs, causing them to be flushed.

The Virtual Addresses held in the MicroTLB include the FCSE translation from *Virtual Address* (VA) to *Modified Virtual Address* (MVA), see the *ARM Architecture Reference Manual Part B*. The process of loading the MicroTLB from the Main TLB includes the FCSE translation if appropriate. The MicroTLB has eight entries.

5.2.2 Main TLB

The Main TLB is the second layer in the TLB structure that catches the cache misses from the MicroTLBs. It provides a centralized source for lockable translation entries.

Misses from the instruction and data MicroTLBs are handled by a unified Main TLB, that is accessed only on MicroTLB misses. Accesses to the Main TLB take a variable number of cycles, according to competing requests between each of the MicroTLBs and other implementation-dependent factors. Entries in the lockable region of the Main TLB are lockable at the granularity of a single entry, as described in *c10, TLB Lockdown Register* on page 3-59.

Main TLB implementation

The Main TLB is implemented as a combination of two elements:

- a fully-associative array of eight elements, which is lockable
- a low-associativity Tag RAM and Data RAM structure similar to that used in the cache.

The implementation of the low-associativity region is a 64-entry 2-way associative structure. Depending on the RAMs available, you can implement this as either:

- four 32-bit wide RAMs
- two 64-bit wide RAMs
- a single 128-bit wide RAM.

Main TLB misses

Main TLB misses are handled in hardware by the level two page table walk mechanism, as used on previous ARM processors. See *c8, TLB Operations Register* on page 3-54.

5.2.3 TLB control operations

The TLB control operations are described in *c8, TLB Operations Register* on page 3-54 and *c10, TLB Lockdown Register* on page 3-59.

5.2.4 Page-based attributes

The page-based attributes for access protection are described in *Memory access control* on page 5-11. The memory types and page-based cache control attributes are described in *Memory region attributes* on page 5-15 and *Memory attributes and types* on page 5-20.

The behavior of memory system when the MMU is disabled is described in *Enabling and disabling the MMU* on page 5-9.

5.2.5 Coherency

Coherency is guaranteed when:

- the cache policy for the page is write-back write-allocate
- the Shared bit is set for the page in the MMU
- the CPU is in coherent mode, that is, bit 5 of the Auxiliary Control Register is set.

When the Shared attribute is applied to a cachable page that does not match these conditions then it is treated as noncachable.

5.2.6 Supersections

In addition to the ARMv6 page types, ARM11 MPCore processors support 16MB pages, which are known as supersections. These are designed for mapping large expanses of the memory map in a single TLB entry.

Supersections are defined using a first level descriptor in the page tables, similar to the way a Section is defined. Because each first level page table entry covers a 1MB region of virtual memory, the 16MB supersections require that 16 identical copies of the first level descriptor of the supersection exist in the first level page table.

Every supersection is defined to have its Domain as 0.

Supersections can be specified regardless of whether subpages are enabled or not, as controlled by the CP15 Control Register XP bit (bit 23). The page table formats of supersections are shown in Figure 5-5 on page 5-43 and Figure 5-9 on page 5-46.

5.3 Memory access sequence

When the ARM11 MPCore processor generates a memory access, the MMU:

1. Performs a lookup for a mapping for the requested Virtual Address and current ASID in the relevant Instruction or Data MicroTLB.
2. If step 1 misses then a lookup for a mapping for the requested Virtual Address and current ASID in the Main TLB is performed.

If no global mapping, or mapping for the currently selected ASID, for the Virtual Address can be found in the TLBs then a translation table walk is automatically performed by hardware. See *Hardware page table translation* on page 5-40.

If a matching TLB entry is found then the information it contains is used as follows:

1. The access permission bits and the domain are used to determine the access privileges for the attempted access. If the privileges are valid the access is enabled to proceed. Otherwise the MMU signals a memory abort. *Memory access control* on page 5-11 describes how this is done.
2. The memory region attributes are used to control the Cache and Write Buffer, and to determine if the access is coherent, cached, uncached, or Device, and if it is Shared, as described in *Memory region attributes* on page 5-15.
3. The physical address is used for any access to external memory to perform tag matching for instruction cache entries and lookups and tag matching for data cache entries.

5.3.1 TLB match process

Each TLB entry contains a Virtual Address, a page size, a physical address, and a set of memory properties. Each is marked as being associated with a particular application space, or as global for all application spaces. Register c13 in CP15 determines the currently selected application space. A TLB entry matches if bits [31:N] of the Virtual Address match, where N is $\log_2$ of the page size for the TLB entry. It is either marked as global, or the *Application Space IDentifier* (ASID) matches the current ASID. The behavior of a TLB if two or more entries match at any time, including global and ASID-specific entries, is Unpredictable. The operating system must ensure that, at most, one TLB entry matches at any time. A TLB can store entries based on the following four block sizes:

Supersections	Consist of 16MB blocks of memory.
Sections	Consist of 1MB blocks of memory.
Large pages	Consist of 64KB blocks of memory.

Small pages Consist of 4KB blocks of memory.

Supersections, sections, and large pages are supported to permit mapping of a large region of memory while using only a single entry in a TLB. If no mapping for an address is found within the TLB, then the translation table is automatically read by hardware and a mapping is placed in the TLB. See *Hardware page table translation* on page 5-40 for more details.

5.4 Enabling and disabling the MMU

You can enable and disable the MMU by writing the M bit, bit 0, of the CP15 Control Register c1. On reset, this bit is cleared to 0, disabling the MMU.

5.4.1 Enabling the MMU

Before you enable the MMU you must:

1. Program all relevant CP15 registers. This includes setting up suitable translation tables in memory.
2. Disable and invalidate the Instruction Cache. You can then re-enable the Instruction Cache when you enable the MMU.

To enable the MMU proceed as follows:

1. Program the Translation Table Base and Domain Access Control Registers.
2. Program first-level and second-level descriptor page tables as required.
3. Enable the MMU by setting bit 0 in the CP15 Control Register c1.

5.4.2 Disabling the MMU

To disable the MMU proceed as follows:

1. Clear bit 2 in the CP15 Control Register c1. The Data Cache must be disabled prior to, or at the same time as the MMU being disabled, by clearing bit 2 of the Control Register.

   Note

   If the MMU is enabled, then disabled, and subsequently re-enabled, the contents of the TLBs are preserved. If these are now invalid, you must invalidate the TLBs before the MMU is re-enabled (see *c8, TLB Operations Register* on page 3-54

2. Clear bit 0 in the CP15 Control Register c1.

When the MMU is disabled, memory accesses are treated as follows:

- All data accesses are treated as Noncachable. The value of the C bit, bit 2, of the CP15 Control Register c1 Should Be Zero.
- All instruction accesses are treated as Cachable if the I bit, bit 12, of the CP15 Control Register c1 is set to 1, and Noncachable if the I bit is set to 0.
- All explicit accesses are Strongly Ordered. The value of the W bit, bit 3, of the CP15 Control Register c1 is ignored.

- No memory access permission checks are performed, and no aborts are generated by the MMU.
- The physical address for every access is equal to its Virtual Address. This is known as a flat address mapping.
- The FCSE PID Should Be Zero when the MMU is disabled. This is the reset value of the FCSE PID. If the MMU is to be disabled the FCSE PID must be cleared.
- All change of program flow prediction is disabled. The state of the Z bit, bit 11, of the CP15 Control Register c1 is ignored. This prevents speculative fetches before the memory region types are defined, protecting read-sensitive I/O locations.
- All CP15 MMU and cache operations work as normal when the MMU is disabled.

5.5 Memory access control

Access to a memory region is controlled by

- *Domains*
- *Access permissions* on page 5-12
- *Execute never bits in the TLB entry* on page 5-13.

5.5.1 Domains

A domain is a collection of memory regions. The ARM architecture supports 16 domains. Domains provide support for multi-user operating systems. All regions of memory have an associated domain.

A domain is the primary access control mechanism for a region of memory and defines the conditions in which an access can proceed. The domain determines whether:

- access permissions are used to qualify the access
- access is unconditionally allowed to proceed
- access is unconditionally aborted.

In the latter two cases, the access permission attributes are ignored.

Each page table entry and TLB entry contains a field that specifies which domain the entry is in. Access to each domain is controlled by a 2-bit field in the Domain Access Control Register, CP15 c3. Each field enables very quick access to be achieved to an entire domain, so that whole memory areas can be efficiently swapped in and out of virtual memory. Two kinds of domain access are supported:

Clients Clients are users of domains in that they execute programs and access data. They are guarded by the access permissions of the TLB entries for that domain.

A client is a domain user, and each access must be checked against the access permission settings for each memory block and the system protection bit, the S bit, and the ROM protection bit, the R bit, in CP15 Control Register c1. Table 5-1 on page 5-12 shows the access permissions.

Managers Managers control the behavior of the domain, the current sections and pages in the domain, and the domain access. They are not guarded by the access permissions for TLB entries in that domain.

Because a manager controls the domain behavior, each access has only to be checked to be a manager of the domain.

One program can be a client of some domains, and a manager of some other domains, and have no access to the remaining domains. This enables flexible memory protection for programs that access different memory resources.

5.5.2 Access permissions

The access permission bits control access to the corresponding memory region. If an access is made to an area of memory without the required permissions, then a permission fault is raised.

The access permissions are determined by a combination of the AP and APX bits in the page table, and the S and R bits in CP15 Control Register c1. For page tables not supporting the APX bit, the value 0 is used.

Changes to the S and R bits do not affect the access permissions of entries already in the TLB. You must flush the TLB to enable the updated S and R bits to take effect.

——— **Note** ———

The use of the S and R bits is deprecated.

———————————

The encoding of the access permission bits is shown in Table 5-1.

Table 5-1 Access permission bits encoding

S	R	APX	AP[1:0]	Privileged permissions	User permissions	Description
0	0	0	b00	No access	No access	All accesses generate a permission fault
x	x	0	b01	R/W	No access	Privileged access only
x	x	0	b10	R/W	RO	Writes in User mode generate permission faults
x	x	0	b11	R/W	R/W	Full access
0	0	1	b00	-	-	Reserved
0	0	1	b01	RO	No access	Privileged RO
0	0	1	b10	RO	RO	Privileged/User RO
0	0	1	b11	RO	RO	Privileged/User RO
The S and R bits are deprecated. The following entries apply to legacy systems only.						
0	1	0	b00	RO	RO	Privileged/User RO
1	0	0	b00	RO	No access	Privileged RO

Table 5-1 Access permission bits encoding (continued)

S	R	APX	AP[1:0]	Privileged permissions	User permissions	Description
1	1	0	b00	-	-	Reserved
0	1	1	xx	-	-	Reserved
1	0	1	xx	-	-	Reserved
1	1	1	xx	-	-	Reserved

5.5.3 Execute never bits in the TLB entry

Each memory region can be tagged as not containing executable code. If the Execute Never, XN, bit of the TLB Attributes Entry Register, CP15 c10, is set to 1, then any attempt to execute an instruction in that region results in a permission fault. If the XN bit is cleared to 0, then code can execute from that memory region. See *ARMv6 page table translation (subpage AP bits disabled)* on page 5-43 for more details.

5.5.4 Access permission and ForceAP bit

APX, AP[1:0] encoding is shown in Table 5-2.

Table 5-2 Access Permission Bits

APX	AP[1:0]	Privileged permissions	User permissions	Description
0	00	No Access	No Access	All accesses generate a permission fault
0	01	R/W	No Access	Privileged access only
0	10	R/W	RO	Writes in user mode generate permission faults
0	11	R/W	R/W	Full access
1	00	---	---	Reserved
1	01	RO	No Access	Privileged RO
1	10	RO	RO	Privileged/User RO
1	11	RO	RO	Privileged/User RO

The AP bits in the v6 MMU specification currently require the following encodings on APX, AP[1:0]:

- RO by both privileged and unprivileged code: b111
- RW by both privileged and unprivileged code: b011
- RO by privileged, NoAccess by unprivileged: b101
- RW by privileged, No Access by unprivileged: b001

So APX becomes RO/not RW and AP[1] becomes User/not Kernel, when AP[0] is set to 1.

Access Bit

The freeing of AP[0] = 0 for the common permission cases above allows the introduction of an Access Bit. The Access Bit records whether a page (or section) has been accessed by the TLB recently, and can be used to optimize memory management algorithms in the OS. The Access Bit is managed by software for MPCore.

Reading a page table entry into the TLB where the access bit is 0 causes a fault that can be quickly distinguished from every other TLB generated fault. This enables fast setting of the Access Bit in software. Functionally, it behaves in a very similar manner to the Translation fault, but avoids the need to distinguish between unallocated entries (handled by the Translation Fault) and entries not recently used (handled by the Access Bit Fault)

If AP[0] is set to 0, then a separate control bit in CP15 Register 1, bit [29] (ForceAP) is used to cause the new Access Bit Fault abort type.

The priority of the Access Bit Fault is between the Translation Fault and the Domain Fault. The TLB entry which cause the Access Bit fault does not need to be flushed from the TLB after setting the Access Bit to 1 after an Access Bit fault

It is Unpredictable whether the TLB caches the effect of the ForceAP bit in CP15 Register 1. As a result, the TLB must be invalidated between changing the ForceAP bit and relying on the effect of those changes taking place.

5.6 Memory region attributes

Each TLB entry has an associated set of memory region attributes. These control:

- accesses to the caches
- how the Write Buffer is used
- if the memory region is shareable and must be kept coherent.

The MP11 CPU data caches are write-back caches only. In addition, only one cache allocation policy is supported, that is, the L1 data cache performs a linefill on every read miss and on every write miss. As a consequence, Inner Write-Through No Allocate on Write behaves as Inner Noncachable and Inner Write-Back No Allocate on Write behaves as Write-back Write-Allocate

5.6.1 C and B bit, and type extension field encodings

Page table formats use five bits to encode the memory region type. These are **TEX[2:0]**, and the C and B bits. Table 5-3 shows the mapping of the *Type Extension Field* (TEX) and the C and B bits to memory region type. For page tables formats with no TEX field you must use the value b000.

Table 5-3 TEX field, and C and B bit encodings used in page table formats

Page table encodings			Description	Memory type	Page shared?
TEX	C	B			
b000	0	0	Strongly Ordered	Strongly Ordered	Shared[a]
b000	0	1	Shared Device	Device	Shared[a]
b000	1	0	Outer and Inner Write-Through, No Allocate on Write	Normal	S[b]
b000	1	1	Outer and Inner Write-Back, No Allocate on Write	Normal	S[b]
b001	0	0	Outer and Inner Noncachable	Normal	S[b]
b001	0	1	Reserved	-	-
b001	1	0	Reserved	-	-
b001	1	1	Outer and Inner Write-Back, Write Allocate	Normal	S[b]

Table 5-3 TEX field, and C and B bit encodings used in page table formats (continued)

Page table encodings			Description	Memory type	Page shared?
TEX	C	B			
b010	0	0	Non-Shared Device	Device	Non-Shared
b010	0	1	Reserved	-	-
b010	1	X	Reserved	-	-
b011	X	X	Reserved	-	-
1BB	A	A	Cached memory. BB = Outer policy, AA = Inner policy. See Table 5-4.	Normal	s[b]

a. Shared, regardless of the value of the S bit in the page table.
b. s is Shared if the value of the S bit in the page table is 1, or Non-Shared if the value of the S bit is 0 or not present.

Additionally certain page tables contain the Shared bit, S. This bit only applies to Normal, not Device or Strongly Ordered memory, and determines if the memory region is Shared (1), or Non-Shared (0). If not present the S bit is assumed to be 0 (Non-Shared).

Despite the fact that the B bit is clear for Non-Shared Device, this is just a reusage of the encoding, and does not imply that the access is not buffered.

The Inner and Outer cache policy bits AA (C and B bits) and BB (TEX[1:0]) control the operation of memory accesses to the external memory. Table 5-4 indicates how the MMU and cache interpret the cache policy bits.

Table 5-4 Cache policy bits

TEX[1:0] (BB) or CB (AA) bits	Cache policy
b00	Noncachable, Unbuffered
b01	Write-Back cached, Write Allocate, Buffered
b10	Write-Through cached, No Allocate on Write, Buffered
b11	Write-Back cached, No Allocate on Write, Buffered

The terms Inner and Outer refer to levels of caches that can be built in a system. Inner refers to the innermost caches, including level one. Outer refers to the outermost caches. The boundary between Inner and Outer caches is defined in the implementation of a cached system. Inner must always include level one. In a system with three levels of caches, an example is for the Inner attributes to apply to level one and level two, while the Outer attributes apply to level three. In a two-level system, it is envisaged that Inner always applies to level one and Outer to level two.

In ARM11 MPCore processors, Inner refers to level one. **ARCACHE** and **AWCACHE** show the Outer Cachable properties. **ARUSER and AWUSER** show the Inner Cachable values.

For an explanation of Strongly Ordered and Device see *Memory attributes and types* on page 5-20.

You can choose which write allocation policy an implementation supports. The Allocate On Write and No Allocate On Write cache policies indicate which allocation policy is preferred for a memory region, but you must not rely on the memory system implementing that policy. ARM11 MPCore processors only support Inner Allocate on Write and do not support No Allocate On Write.

Not all Inner and Outer cache policies are mandatory. Table 5-5 shows the possible implementation options.

Table 5-5 Inner and Outer cache policy implementation options

Cache policy	Implementation options	Supported by ARM11 MPCore processors?
Inner Noncachable	Mandatory.	Yes
Inner Write-Through	Mandatory.	Yes, behaves as noncachable
Inner Write-Back	Optional. If not supported, the memory system must implement this as Inner Write-Through.	Yes, all treated as write-back write-allocate
Outer Noncachable	Mandatory.	System-dependent
Outer Write-Through	Optional. If not supported, the memory system must implement this as Outer Noncachable.	System-dependent
Outer Write-Back	Optional. If not supported, the memory system must implement this as Outer Write-Through.	System-dependent

5.6.2 Shared

This bit indicates that the memory region can be shared by multiple processors. For a full explanation of the Shared attribute see *Memory attributes and types* on page 5-20.

5.6.3 Page table descriptors when using remapping

Table 5-3 on page 5-15 shows the existing V6 TEX with CB encodings. This can be re-arranged to provide a mechanism to indirect the functions of the encodings TEX[0], C and B provide, using remap registers. TEX[2:1] are then freed up for software usage. This behavior is only enabled when CP15 register 1, bit [28] is set, so providing backward compatibility with ARMv6.

The result is then that if CP15 Register 1, bit [28] is set, the effect is as shown in Table 5-6.

Table 5-6 New V6 TEX, CB encodings

Page Table Encodings			Memory Type	Inner Cache Attributes when mapped as Normal	Outer Cache Attributes when mapped as Normal
TEX	C	B			
XX0	0	0	PRRR[1:0]	NMRR[1:0]	NMRR[17:16]
XX0	0	1	PRRR[3:2]	NMRR[3:2]	NMRR[19:18]
XX0	1	0	PRRR[5:4]	NMRR[5:4]	NMRR[21:20]
XX0	1	1	PRRR[7:6]	NMRR[7:6]	NMRR[23:22]
XX1	0	0	PRRR[9:8]	NMRR[9:8]	NMRR[25:24]
XX1	0	1	PRRR[11:10]	NMRR[11:10]	NMRR[27:26]
XX1	1	0	RESERVED	RESERVED	RESERVED
XX1	1	1	PRRR[15:14]	NMRR[15:14]	NMRR[31:30]

Primary memory region remap registers (PRRR) and normal memory region remap registers (NMRR) are defined in *c10, Memory remap registers* on page 3-60.

Table 5-7 Page attributes and memory types

Memory Type	Shareable attribute when S = 0	Shareable attribute when S = 1
Strongly Ordered	Shareable	Shareable
Device	PRRR[16]	PRRR[17]
Normal	PRRR[18]	PRRR[19]

To provide different encodings of the Outer cache attributes from the Inner cache attributes, the MMU remap register allows the cachable attributes to be remapped to different values.

It is Unpredictable whether the TLB caches the effect of the TEX Remap on page tables. As a result, the TLB must be invalidated between changing the TEX Remap bit and relying on the effect of those changes taking place.

5.7 Memory attributes and types

The ARM11 MPCore processor provides a set of memory attributes that have characteristics that are suited to particular devices, including memory devices, that can be contained in the memory map. The ordering of accesses for regions of memory is also defined by the memory attributes. There are three mutually exclusive main memory type attributes:

- Strongly Ordered
- Device
- Normal.

These are used to describe the memory regions. The marking of the same memory locations as having two different attributes in the MMU, for example using synonyms in a virtual to physical address mapping, results in Unpredictable behavior.

A summary of the memory attributes is shown in Table 5-8.

Table 5-8 Memory attributes

Memory type attribute	Shared/ Non-Shared	Other attributes	Description
Strongly Ordered	-	-	All memory accesses to Strongly Ordered memory occur in program order. Some backwards compatibility constraints exist with ARMv5 instructions that change the CPSR interrupt masks (see *Strongly Ordered memory attribute* on page 5-24). All Strongly Ordered accesses are assumed to be shared.
Device	Shared	-	Designed to handle memory-mapped peripherals that are shared by several processors.
	Non-Shared	-	Designed to handle memory-mapped peripherals that are used only by a single processor.

Table 5-8 Memory attributes

Memory type attribute	Shared/ Non-Shared	Other attributes	Description
Normal	Shared	Write-Back Cachable	Coherent mode handled by the ARM11 MPCore processor. Used with the SMP bit of the Auxiliary Control Register. See *AMP mode and SMP mode* on page 3-35.
		Noncachable/ Write-Through Cachable (treated as Noncachable)	Designed to handle normal memory that is shared between several cores in a system. No data coherency is maintained between MP11 CPUs level 1 data caches within the ARM11 MPCore processor.
	Non-Shared	Noncachable/ Write-Through Cachable (treated as Noncachable)/ Write-Back Cachable	Designed to handle normal memory that is used only by a single processor.

5.7.1 Normal memory attribute

The Normal memory attribute is defined on a per-page basis in the MMU and provides memory access orderings that are suitable for normal memory. This type of memory stores information without side effects. Normal memory can be writable or read-only.

For writable normal memory, unless there is a change to the physical address mapping:

- a load from a specific location returns the most recently stored data at that location for the same processor
- two loads from a specific location, without a store in between, return the same data for each load.

For read-only normal memory:

- two loads from a specific location return the same data for each load.

This behavior describes most memory used in a system, and the term memory-like is used to describe this sort of memory. In this section, writable normal memory and read-only normal memory are not distinguished.

Regions of memory with the Normal attribute can be Shared or Non-Shared, on a per-page basis in the MMU. The marking of the same memory locations as being Shared Normal and Non-Shared Normal in the MMU, for example by the use of synonyms in a virtual to physical address mapping, results in Unpredictable behavior.

All explicit accesses to memory marked as Normal must correspond to the ordering requirements of accesses described in *Ordering requirements for memory accesses* on page 5-25. Accesses to Normal memory conform to the Weakly Ordered model of memory ordering. A description of this model is in standard texts describing memory ordering issues.

Shared Normal memory

The Shared Normal memory attribute is designed to describe normal memory that can be accessed by multiple processors or other system masters.

A region of memory marked as Shared Normal is one in which the effect of interposing a cache, or caches, on the memory system is entirely transparent. Implementations can use a variety of mechanisms to support this, from not caching accesses in shared regions to more complex hardware schemes for cache coherency for those regions. ARM11 MPCore processors cache shareable locations when the following conditions are met:

- the cache policy for the page is write-back write-allocate
- the Shared bit is set for the page in the MMU
- the MP11 CPU is in coherent mode, that is, bit 5 of the Auxiliary Control Register is set
- the SCU has been turned on to maintain coherency in the ARM11 MPCore processor (see *SCU Control Register* on page 9-4).

Note

When the conditions are met the ARM11 MPCore processor only guarantees coherency among the L1 memory of its MP11 CPUs. Accesses by other system masters without software support for coherency have unpredictable results.

When the Shared attribute is applied to a cachable page that does not match these conditions then it is treated as noncachable.

Non-Shared Normal memory

The Non-Shared Normal memory attribute describes normal memory that can be accessed only by a single processor.

A region of memory marked as Non-Shared Normal does not have any requirement to make the effect of a cache transparent.

Cachable Write-Through, Cachable Write-Back, and Noncachable

In addition to marking a region of Normal memory as being Shared or Non-Shared, a region of memory marked as Normal can also be marked on a per-page basis in an MMU as being one of:

- Cachable Write-Through is mapped to Noncachable as shown in Table 5-5 on page 5-17
- Cachable Write-Back
- Noncachable.

This marking is independent of the marking of a region of memory as being Shared or Non-Shared, and indicates the required handling of the data region for reasons other than those to handle the requirements of shared data.

The marking of the same memory locations as having different Cachable attributes, for example by the use of synonyms in a virtual to physical address mapping, results in Unpredictable behavior.

5.7.2 Device memory attribute

The Device memory attribute is defined for memory locations where an access to the location can cause side effects, or where the value returned for a load can vary depending on the number of loads performed. Memory-mapped peripherals and I/O locations are typical examples of areas of memory that you must mark as Device. The marking of a region of memory as Device is performed on a per-page basis in the MMU.

Accesses to memory-mapped locations that have side effects that apply to memory locations that are Normal memory might require memory barriers to ensure correct execution. An example where this might be an issue is the programming of the control registers of a memory controller while accesses are being made to the memories controlled by the controller.

Instruction fetches must not be performed to areas of memory containing read-sensitive devices, because there is no ordering requirement between instruction fetches and explicit accesses. As a result, instruction fetches from such devices can result in Unpredictable behavior. Up to 64 bytes can be prefetched sequentially ahead of the current instruction being executed. To enable this, read-sensitive devices must be located in memory in such a way to allow for this prefetching.

Explicit accesses from the processor to regions of memory marked as Device occur at the size and order defined by the instruction. The number of location accesses is specified by the program. Repeat accesses to such locations when there is only one access in the program, that is the accesses are not restartable, are not possible in the

ARM11 MPCore processor. An example of where a repeat access might be required is before and after an interrupt to enable the interrupt to abandon a slow access. You must ensure these optimizations are not performed on regions of memory marked as Device.

If a memory operation that causes multiple transactions (such as an LDM or an unaligned memory access) crosses a 1KB address boundary, then it can perform more accesses than are specified by the program, regardless of one or both of the areas being marked as Device. For this reason, accesses to volatile memory devices must not be made using single instructions that cross a 1KB address boundary. This restriction is expected to cause restrictions to the placing of such devices in the memory map of a system, rather than to cause a compiler to be aware of the alignment of memory accesses.

In addition, address locations marked as Device are not held in a cache.

Shared memory attribute

Regions of memory marked as Device are further distinguished by the Shared attribute in the MMU. These memory regions can be marked as:

- Shared Device
- Non-Shared Device.

Explicit accesses to memory with each of the sets of attributes occur in program order relative to other explicit accesses to the same set of attributes.

All explicit accesses to memory marked as Device must correspond to the ordering requirements of accesses described in *Ordering requirements for memory accesses* on page 5-25.

The marking of the same memory location as being Shared Device and Non-Shared Device in an MMU, for example by the use of synonyms in a virtual to physical address mapping, results in Unpredictable behavior.

For Shared Device memory, the data of a write is visible to all observers in the ARM11 MPCore processor before the end of a Data Synchronization Barrier memory barrier.

5.7.3 Strongly Ordered memory attribute

A further memory attribute, Strongly Ordered, is defined on a per-page basis in the MMU. Accesses to memory marked as Strongly Ordered have a strong memory-ordering model with respect to all explicit memory accesses from that processor. An access to memory marked as Strongly Ordered acts as a *Data Memory Barrier* (DMB) to all other explicit accesses from that processor, until the point at which the access is complete (that is, has changed the state of the target location or data has been returned). In addition, an access to memory marked as Strongly Ordered must complete before the end of a memory barrier (see *Explicit memory barriers* on page 5-28).

To maintain backwards compatibility with ARMv5 architecture, any ARMv5 instructions that implicitly or explicitly change the interrupt masks in the CPSR that appear in program order after a Strongly Ordered access must wait for the Strongly Ordered memory access to complete. These instructions are MSR with the control field mask bit set, and the flag setting variants of arithmetic and logical instructions whose destination register is r15, which copies the SPSR to CPSR. This requirement exists only for backwards compatibility with previous versions of the ARM architecture, and the behavior is deprecated in ARMv6. Programs must not rely on this behavior, but instead include an explicit memory barrier (see *Explicit memory barriers* on page 5-28) between the memory access and the following instruction.

The ARM11 MPCore processor does not require an explicit memory barrier in this situation, but for future compatibility it is recommended that programmers insert a memory barrier.

Explicit accesses from the processor to memory marked as Strongly Ordered occur at their program size, and the number of accesses that occur to such locations is the number that are specified by the program. Implementations must not repeat accesses to such locations when there is only one access in the program (that is, the accesses are not restartable).

If a memory operation that causes multiple transactions (such as LDM or an unaligned memory access) crosses a 4KB address boundary, then it might perform more accesses than are specified by the program regardless of one or both of the areas being marked as Strongly Ordered. For this reason, it is important that accesses to volatile memory devices are not made using single instructions that cross a 4KB address boundary.

Address locations marked as Strongly Ordered are not held in a cache, and are treated as Shared memory locations.

For Strongly Ordered memory, the data and side effects of a write are visible to all observers before the end of a Data Synchronization Barrier memory barrier (see *Explicit memory barriers* on page 5-28).

5.7.4 Ordering requirements for memory accesses

The various memory types defined in this section have restrictions in the memory orderings that are allowed.

Ordering requirements for two accesses

The order of any two explicit architectural memory accesses where one or more are to memory marked as Non-Shared must obey the ordering requirements shown in Table 5-9.

Table 5-9 shows the memory ordering between two explicit accesses A1 and A2, where A1 occurs before A2 in program order.

The symbols used in the table are as follows:

<	Accesses must occur strictly in program order. That is, A1 must occur strictly before A2. It must be impossible to tell otherwise from observation of the read/write values and side effects caused by the memory accesses.
?	Accesses can occur in any order, provided that the requirements of uniprocessor semantics are met, for example respecting dependencies between instructions within a single processor.

Table 5-9 Memory ordering restrictions

A2							
Normal read	Device read (Non-Shared)	Device read (Shared)	Strongly Ordered read	Normal write	Device write (Non-Shared)	Device write (Shared)	Strongly Ordered write

Table 5-9 Memory ordering restrictions

		A2							
A1	Normal read	?	?	?	<	?	?	?	<
	Device read (Non-Shared)	?	<	?	<	?	<	?	<
	Device read (Shared)	?	?	<	<	?	?	<	<
	Strongly Ordered read	<	<	<	<	<	<	<	<
	Normal write	?	?	?	<	?	?	?	<
	Device write (Non-Shared)	?	<	?	<	?	<	?	<
	Device write (Shared)	?	?	<	<	?	?	<	<
	Strongly Ordered write	<	<	<	<	<	<	<	<

There are no ordering requirements for implicit accesses to any type of memory.

Definition of program order of memory accesses

The program order of instruction execution is defined as the order of the instructions in the control flow trace.

Two explicit memory accesses in an execution can either be:

Ordered	Denoted by <. If the accesses are Ordered, then they must occur strictly in order.
Weakly Ordered	Denoted by <=. If the accesses are Weakly Ordered, then they must occur in order or simultaneously.

The rules for determining this for two accesses A1 and A2 are:

1. If A1 and A2 are generated by two different instructions, then:
    - A1 < A2 if the instruction that generates A1 occurs before the instruction that generates A2 in program order.
    - A2 < A1 if the instruction that generates A2 occurs before the instruction that generates A1 in program order.

2. If A1 and A2 are generated by the same instruction, then:
   - If A1 and A2 are the load and store generated by a SWP or SWPB instruction, then:
     - — A1 < A2 if A1 is the load and A2 is the store
     - — A2 < A1 if A2 is the load and A1 is the store.
   - If A1 and A2 are two word loads generated by an LDC, LDRD, or LDM instruction, or two word stores generated by an STC, STRD, or STM instruction, but excluding LDM or STM instructions whose register list includes the PC, then:
     - — A1 <= A2 if the address of A1 is less than the address of A2
     - — A2 <= A1 if the address of A2 is less than the address of A1.
   - If A1 and A2 are two word loads generated by an LDM instruction whose register list includes the PC or two word stores generated by an STM instruction whose register list includes the PC, then the program order of the memory operations is not defined.

Multiple load and store instructions (such as LDM, LDRD, STM, and STRD) generate multiple word accesses, each being a separate access to determine ordering.

5.7.5 Explicit memory barriers

Two explicit memory barrier operations are described in this section:
- Data Memory Barrier
- Data Synchronization Barrier.

In addition, to ensure correct operation where the processor writes code, an explicit Flush Prefetch Buffer operation is provided.

These operations are implemented by writing to the CP15 Cache Operations Register c7. For details of how to use this register see *c7, Cache Operations Register* on page 3-46.

Data Memory Barrier

This memory barrier ensures that all explicit memory transactions occurring in program order before this instruction are completed. No explicit memory transactions occurring in program order after this instruction are started until this instruction completes. Other instructions can complete out of order with the Data Memory Barrier instruction.

Data Synchronization Barrier

This memory barrier completes when all explicit memory transactions occurring in program order before this instruction are completed. No explicit memory transactions occurring in program order after this instruction are started until this instruction completes. In fact, no instructions occurring in program order after the Data Synchronization Barrier complete, or change the interrupt masks, until this instruction completes.

For Shared Device and Normal memory, the data of a write is visible to all observers before the end of a Data Synchronization Barrier. For Strongly Ordered memory, the data and the side effects of a write are visible to all observers before the end of a Data Synchronization Barrier. For Non-Shared Device and Normal memory, the data of a write is visible to the processor before the end of a Data Synchronization Barrier.

Flush Prefetch Buffer

The Flush Prefetch Buffer instruction flushes the pipeline in the processor, so that all instructions following the pipeline flush are fetched from memory, (including the instruction cache), after the instruction has been completed. Combined with Data Synchronization Barrier, and potentially a memory barrier, this ensures that any instructions written by the processor are executed. This guarantee is required as part of the mechanism for handling self-modifying code. The execution of a Data Synchronization Barrier instruction and the invalidation of the Instruction Cache and Branch Target Cache are also required for the handling of self-modifying code.

The Flush Prefetch Buffer is guaranteed to perform this function, while alternative methods of performing the same task, such as a branch instruction, can be optimized in the hardware to avoid the pipeline flush (for example, by using a branch predictor).

Memory synchronization primitives

Memory synchronization primitives, such as Data Synchronization Barrier, exist to ensure synchronization between different processes, which might be running on the same processor or on different processors. You can use memory synchronization primitives in regions of memory marked as Shared and Non-Shared when the processes to be synchronized are running on the same processor. You must only use them in Shared areas of memory when the processes to be synchronized are running on different processors.

5.7.6 Backwards compatibility

The ARMv6 memory attributes are significantly different from those in previous versions of the architecture. Table 5-10 shows the interpretation of the earlier memory types in the light of this definition.

Table 5-10 Memory region backwards compatibility

Previous architectures	ARMv6 attribute
NCNB (Noncachable, Non Bufferable)	Strongly Ordered
NCB (Noncachable, Bufferable)	Shared Device
Write-Through Cachable, Bufferable	Non-Shared Normal (Write-Through Cachable)
Write-Back Cachable, Bufferable	Non-Shared Normal (Write-Back Cachable)

5.8 MMU aborts

Mechanisms that can cause the ARM11 MPCore processor to take an exception because of a memory access are:

MMU fault	The MMU detects a restriction and signals the processor.
Debug abort	Monitor debug-mode debug is enabled and a breakpoint or a watchpoint has been detected.
External abort	The external memory system signals an illegal or faulting memory access.

Collectively these are called *aborts*. Accesses that cause aborts are said to be aborted. If the memory request that aborts is an instruction fetch, then a Prefetch Abort exception is raised if and when the processor attempts to execute the instruction corresponding to the aborted access.

If the aborted access is a data access or a cache maintenance operation, a Data Abort exception is raised.

All Data Aborts, and aborts caused by cache maintenance operations, cause the *Data Fault Status Register* (DFSR) to be updated so that you can determine the cause of the abort.

For all aborts, excluding External Aborts, other than on translation, the *Fault Address Register* (FAR) is updated with the address that caused the abort. External Data Aborts, other than on translation, can all be imprecise and therefore the FAR does not contain the address of the abort. See *Imprecise Data Abort mask in the CPSR/SPSR* on page 2-31 for more details on imprecise Data Aborts.

For instruction aborts the value of r14 is used by the abort handler to determine the address that caused the abort.

5.8.1 External aborts

External memory errors are defined as those that occur in the memory system other than those that are detected by an MMU. External memory errors are expected to be extremely rare and are likely to be fatal to the running process. An example of an event that can cause an external memory error is an uncorrectable parity or ECC failure on a level two memory structure.

External abort on instruction fetch

Externally generated errors during an instruction prefetch are precise in nature, and are only recognized by the processor if it attempts to execute the instruction fetched from the location that caused the error. The resulting failure is reported in the Instruction Fault Status Register if no higher priority abort (including a Data Abort) has taken place.

If there is an External Abort during a cache line fill to the a memory barrier, the cache line being filled is not marked as valid. If the abort occurred on a word that the core subsequently attempts to execute, a precise abort occurs.

The Fault Address Register is not updated on an External Abort on instruction fetch.

External abort on data read/write

Externally generated errors during a data read or write can be imprecise. This means that r14_abt on entry into the abort handler on such an abort might not hold an address that is related to the instruction that caused the exception. Correspondingly, External Aborts can be unrecoverable. See *Aborts* on page 2-29 for more details.

If there is an External Abort during a cache line fill to the data cache, the cache line being filled is not marked as valid. If the abort occurred on a word that the core has requested, then the core takes an External Abort.

This abort can be precise or imprecise. See *External Aborts handling* on page 7-5.

The Fault Address Register is not updated on an imprecise External Abort on a data access.

External abort on a hardware page table walk

An External Abort occurring on a hardware page table access must be returned with the page table data. Such aborts are precise. The Fault Address Register is updated on an External Abort on a hardware page table walk on a data access, but not on an instruction access. The appropriate Fault Status Register indicates that this has occurred.

5.9 MMU fault checking

During the processing of a section or page, the MMU behaves differently because it is checking for faults. The MMU generates four types of fault:

- *Alignment fault* on page 5-36
- *Translation fault* on page 5-36
- *Domain fault* on page 5-37
- *Permission fault* on page 5-37.

Aborts that are detected by the MMU are taken before any external memory access takes place.

Alignment fault checking is enabled by the A bit in the Control Register CP15 c1. Alignment fault checking is independent of the MMU being enabled. Translation, domain, and permission faults are only generated when the MMU is enabled.

The access control mechanisms of the MMU detect the conditions that produce these faults. If a fault is detected as the result of a memory access, the MMU aborts the access and signals the fault condition to the processor. The MMU retains status and address information about faults generated by data accesses in DFSR and FAR (see *Fault status and address* on page 5-38). The MMU does not retain status about faults generated by instruction fetches.

An access violation for a given memory access inhibits any corresponding external access, and an abort is returned to the ARM11 MPCore processor.

5.9.1 Fault checking sequence

Figure 5-1 on page 5-34 and Figure 5-2 on page 5-35 show the fault checking sequence for translation table managed TLB modes.

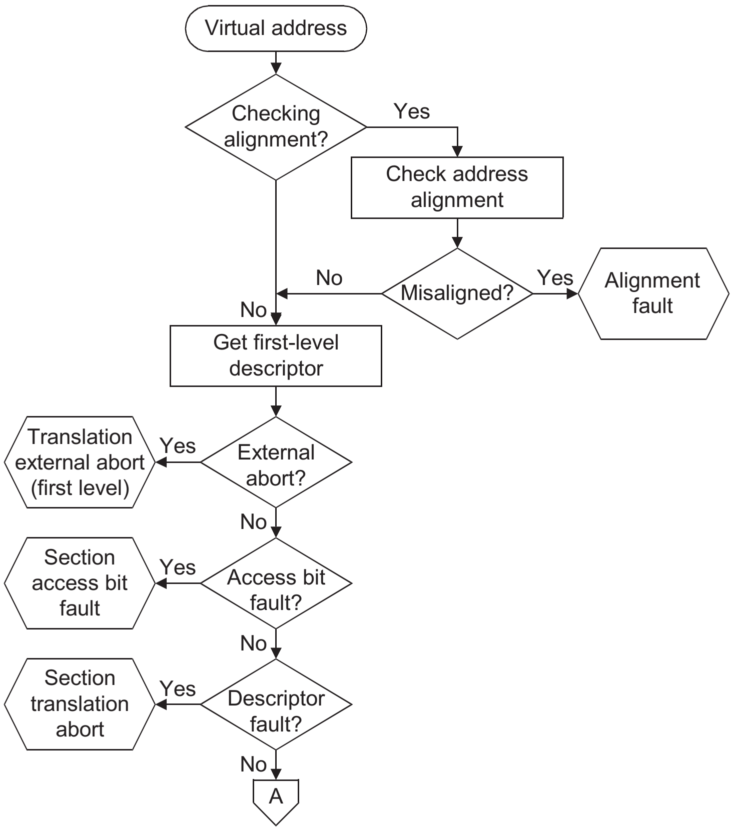


Figure 5-1 Translation table managed TLB fault checking sequence 1

Figure 5-2 on page 5-35 shows the fault checking sequence for translation table managed TLB modes.

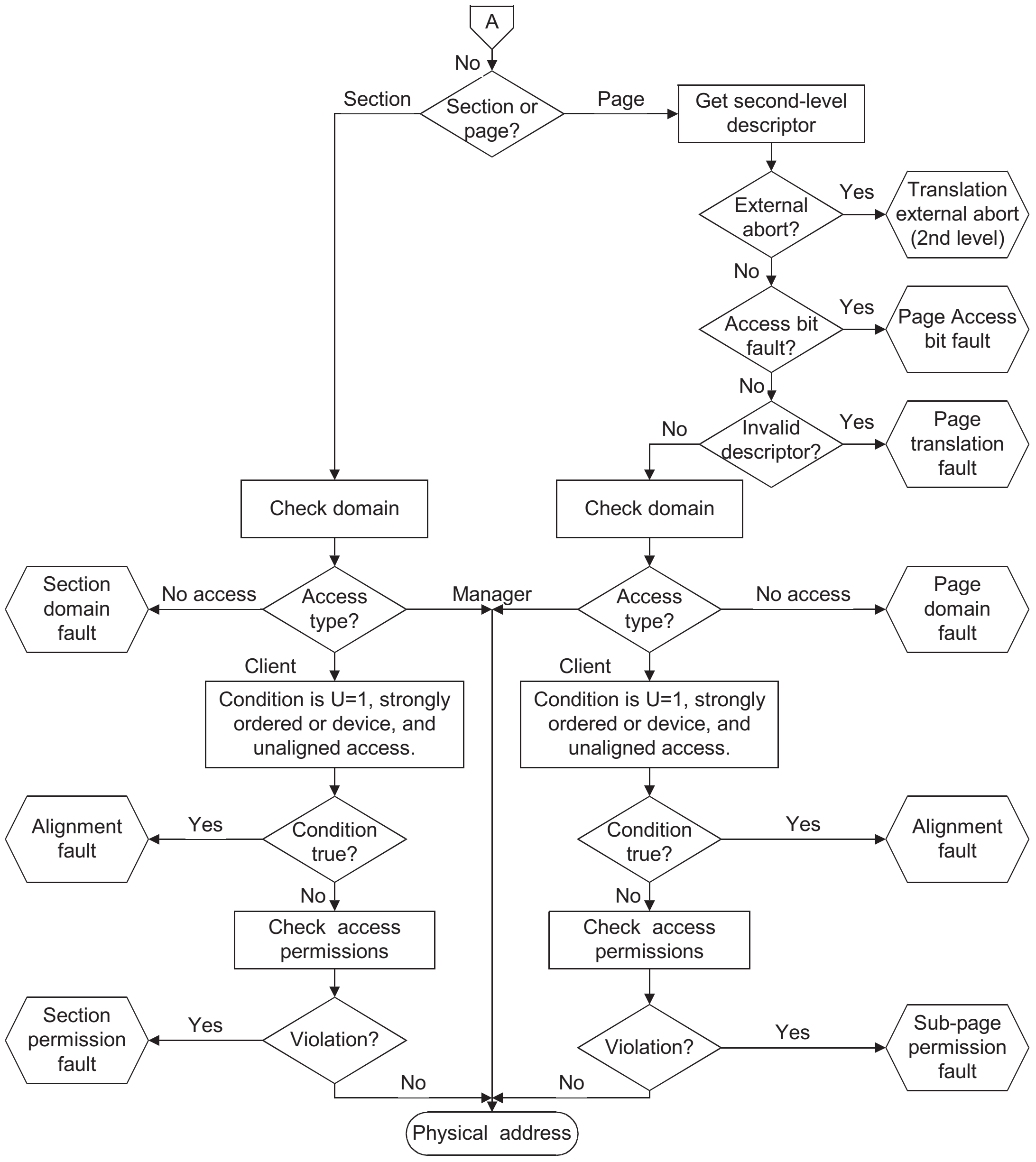


Figure 5-2 Translation table managed TLB fault checking sequence 2

5.9.2 Alignment fault

An alignment fault occurs if the ARM11 MPCore processor has attempted to access a particular data memory size at an address location that is not aligned with that size.

Alignment checks are performed with the MMU both enabled and disabled.

5.9.3 Translation fault

There are two types of translation fault:

Section	A section translation fault occurs if the first-level translation table descriptor is marked as invalid, bits [1:0] = b00.
Page	A page translation fault occurs if the second-level translation table descriptor is marked as invalid, bits [1:0] = b00.

5.9.4 Access bit fault

This bit is only taken into account when the MMU is in ARMv6 mode, that is XP=1, bit [23] in the CP15 Control register.

In the configuration XP=1 and ForceAP=1, the OS uses only bits APX and AP[1] as Access Permission bits, and AP[0] becomes an Access Bit, see *Access permission and ForceAP bit* on page 5-13. The Access Bit records recent TLB access to a page, or section, and the OS can use this to optimize memory managements algorithms.

In the ARM11 MPCore processor the Access Bit must be managed by the software.

Reading a page table entry into the TLB when the Access Bit is 0 causes an Access Bit fault. This fault is readily distinguished from other faults that the TLB generates and this permits fast setting of the Access Bit in software.

The processor can generate two kind of Access Bit faults:

- Section Access Bit fault, when the Access Bit, AP[0], is contained in a first level translation table descriptor
- Page Access Bit fault, when the Access Bit, AP[0], is contained in a second level translation table descriptor

The Force AP and XP bits are expected to be static throughout operations.

It is Unpredictable whether the TLB caches the effect of the Force AP bit on page tables. As a result, the TLB must be invalidated between changing the Force AP bit and relying on the effect of those changes taking place.

5.9.5 Domain fault

There are two types of domain fault:

Section	For a section the domain is checked when the first-level descriptor is returned.
Page	For a page the domain is checked when the second-level descriptor is returned.

For each type, the first-level descriptor indicates the domain in CP15 c3, the Domain Access Control Register, to select. If the selected domain has bit 0 set to 0 indicating either no access or reserved, then a domain fault occurs.

5.9.6 Permission fault

If the two-bit domain field returns Client, the access permission check is performed on the access permission field in the TLB entry. A permission fault occurs if the access permission check fails.

5.9.7 Debug event

When Monitor debug-mode debug is enabled an abort can be taken caused by a breakpoint on an instruction access or a watchpoint on a data access. In both cases the memory system completes the access before the abort is taken. If an abort is taken when in Monitor debug-mode debug then the appropriate FSR (IFSR or DFSR) is updated to indicate a debug abort.

If a watchpoint is taken the WFAR is set to the address that caused the watchpoint. Watchpoints are not taken precisely because following instructions can run underneath load and store multiples. The debugger must read the WFAR to determine which instruction caused the debug event.

5.10 Fault status and address

Table 5-11 shows the encodings for the Fault Status Register.

Table 5-11 Fault Status Register encoding

Priority	Sources		FSR[10,3:0]	Domain	FAR	SLVER/ DECERR	R/W
Highest	Alignment		b00001	Invalid	Valid	Invalid	Valid
	Instruction cache maintenance[a] operation fault		b00100	Invalid	Valid	Invalid	Invalid
	External abort on translation	First-level	b01100	Invalid	Valid	Valid	Valid
		Second-level	b01110	Valid	Valid	Valid	Valid
	Translation	Section	b00101	Invalid	Valid	Invalid	Valid
		Page	b00111	Valid	Valid	Invalid	Valid
	Access bit	Section	b00011	Invalid	Valid	Invalid	Valid
		Page	b00110	Valid	Valid	Invalid	Valid
	Domain	Section	b01001	Valid	Valid	Invalid	Valid
		Page	b01011	Valid	Valid	Invalid	Valid
	Permission	Section	b01101	Valid	Valid	Invalid	Valid
		Page	b01111	Valid	Valid	Invalid	Valid
	Precise External Abort	-	b01000	Valid	Valid	Invalid	Valid
	Imprecise External Abort		b10110	Invalid	Invalid	Valid	Valid
Lowest	Debug event		b00010	Valid	Invalid	Invalid	Invalid

a. These aborts cannot be signaled with the IFSR because they do not occur on the instruction side.

Note

All other Fault Status Register encodings are reserved.

If a translation abort occurs during a Data Cache maintenance operation by Virtual Address, then a Data Abort is taken and the DFSR indicates the reason. The FAR indicates the faulting address.

If a translation abort occurs during an Instruction cache maintenance operation by Virtual Address, then a Data Abort is taken, and an Instruction cache maintenance operation fault is indicated in the DFSR. The FAR indicates the faulting address.

Domain and fault address information is only available for data accesses. For instruction aborts r14 must be used to determine the faulting address. You can determine the domain information by performing a TLB lookup for the faulting address and extracting the domain field.

Table 5-12 is a summary of which abort vector is taken, and which of the Fault Status and Fault Address Registers are updated for each abort type.

Table 5-12 Summary of aborts

Abort type	Abort taken	Precise?	Register updated?			
			IFSR	WFAR	DFSR	FAR
Instruction MMU fault	Prefetch Abort	Yes	Yes	No	No	No
Instruction debug abort	Prefetch Abort	Yes	Yes	No	No	No
Instruction External Abort on translation	Prefetch Abort	Yes	Yes	No	No	No
Instruction External Abort	Prefetch Abort	Yes	Yes	No	No	No
Memory barrier maintenance operation	Data Abort	Yes	Yes	Yes[a]	Yes	Yes
Data MMU fault	Data Abort	Yes	No	Yes[a]	Yes	Yes
Data debug abort	Data Abort	No	No	Yes	Yes	Yes[b]
Data External Abort on translation	Data Abort	Yes	No	Yes[a]	Yes	Yes
Data External Abort	Data Abort	No[c]	No	No	Yes	No
Data cache maintenance operation	Data Abort	Yes	No	Yes[a]	Yes	Yes

a. Although the WFAR is updated by the processor the behavior is architecturally Unpredictable.
b. The processor updates the FAR with an Unpredictable value.
c. Data Aborts can be precise, see *External aborts* on page 5-31 for more details.

5.11 Hardware page table translation

The ARM11 MPCore processor MMU implements the hardware page table walking mechanism from ARMv4 and ARMv5 cached processors with the exception of the fine page table descriptor.

A hardware page table walk occurs whenever there is a TLB miss. MPCore hardware page table walks do not cause a read from the level one Unified/Data Cache. The P, RGN, S, and C bits in the Translation Table Base Registers determine the memory region attributes for the page table walk.

Two formats of page tables are supported:

- A backwards-compatible format supporting subpage access permissions. These have been extended so that certain page table entries support extended region types.
- ARMv6 format, not supporting sub-page access permissions, but with support for ARMv6 MMU features. These features are:
  - extended region types
  - global and process specific pages
  - more access permissions
  - marking of Shared and Non-Shared regions
  - marking of Execute-Never regions.

Additionally two translation table base registers are provided. On a TLB miss, the Translation Table Base Control Register, CP15 c2, and the top bits of the Virtual Address determine if the first or second translation table base is used. See *c2, Translation Table Base Control Register* on page 3-39 for details. The first-level descriptor indicates whether the access is to a section or to a page table. If the access is to a page table, the MPCore MMU fetches a second-level descriptor.

A page table holds 256 32-bit entries 4KB in size. You can determine the page type by examining bits [1:0] of the second-level descriptor.

For both first and second level descriptors if bits [1:0] are b00, the associated Virtual Addresses are unmapped, and attempts to access them generate a translation fault. Software can use bits [31:2] for its own purposes in such a descriptor, because they are ignored by the hardware. Where appropriate, ARM Limited recommends that bits [31:2] continue to hold valid access permissions for the descriptor.

5.11.1 Backwards-compatible page table translation (subpage AP bits enabled)

When the CP15 Control Register c1 bit 23 is set to 0, the subpage AP bits are enabled and the page table formats are backwards-compatible with ARMv4 and ARMv5 MMU architectures.

All mappings are treated as global, and executable (XN = 0). All Normal memory is Non-Shared. Device memory can be Shared or Non-Shared as determined by the TEX bits and the C and B bits.

For large and small pages, there can be four subpages defined with different access permissions. For a large page, the subpage size is 16KB and is accessed using bits [15:14] of the page index of the Virtual Address. For a small page, the subpage size is 1KB and is accessed using bits [11:10] of the page index of the Virtual Address.

The use of subpage AP bits where AP3, AP2, AP1, and AP0 contain different values is deprecated.

Caution

1KB subpages are supported but deprecated. 1KB data subpages are not allocated in the Data MicroTLB. Using 1KB data subpages decreases performance.

Backwards-compatible page table format

Figure 5-3 shows a backwards-compatible format first-level descriptor.

	31 24	23 20	19	18	17 15	14 12	11 10	9	8 5	4	3	2	1	0
Translation fault	Ignored												0	0
Coarse page table	Coarse page table base address							P	Domain	SBZ			0	1
Section (1MB)	Section base address		SBZ	0	SBZ	TEX	AP	P	Domain	0	C	B	1	0
Supersection (16MB)	Supersection base address	SBZ		1	SBZ	TEX	AP	P	Ignored	0	C	B	1	0
Reserved													1	1

Figure 5-3 Backwards-compatible first-level descriptor format

If the P bit is supported and set for the memory region, it indicates to the system memory controller that this memory region has ECC enabled.

Figure 5-4 shows a backwards-compatible format second-level descriptor for a page table.

	31–16	15–12	11–10	9	8	7–6	5–4	3	2	1	0
Translation fault	Ignored									0	0
Large page (64KB)	Large page table base address	TEX	AP3	AP2		AP1	AP0	C	B	0	1
Small page (4KB)	Small page table base address		AP3	AP2		AP1	AP0	C	B	1	0
	Extended small page table base address		SBZ		TEX		AP	C	B	1	1

Figure 5-4 Backwards-compatible second-level descriptor format

For extended small page table entries without a TEX field you must use the value b000.

For details of TEX encodings see *C and B bit, and type extension field encodings* on page 5-15.

Figure 5-5 on page 5-43 shows an overview of the section, supersection, and page translation process using backwards-compatible descriptors.

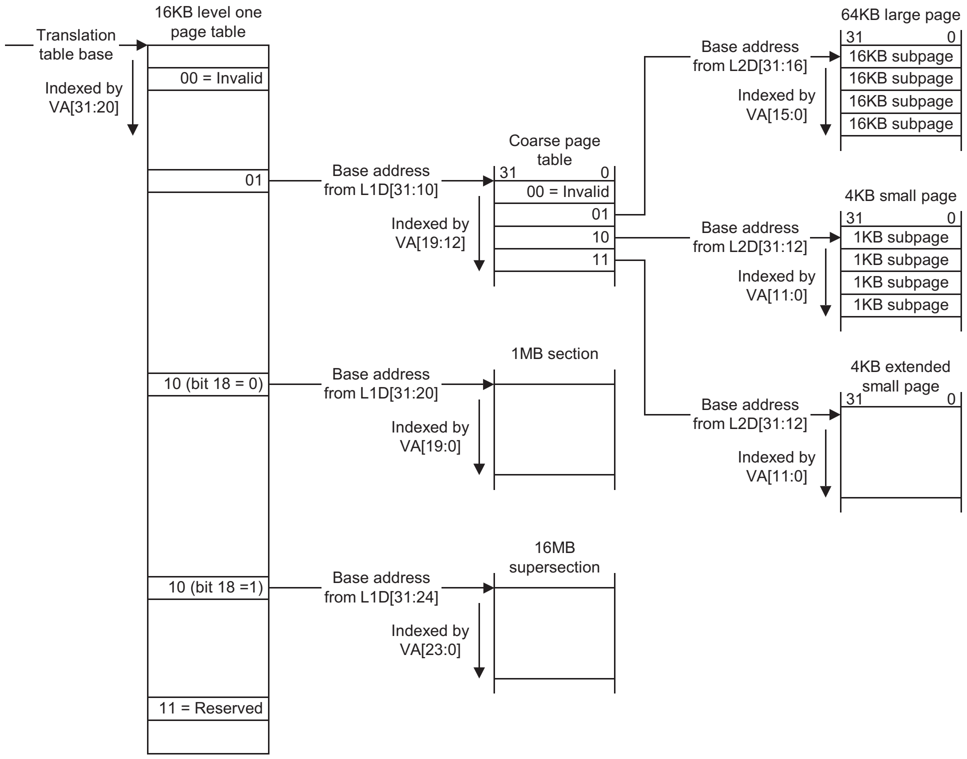


Figure 5-5 Backwards-compatible section, supersection, and page translation

5.11.2 ARMv6 page table translation (subpage AP bits disabled)

When the CP15 Control Register c1 Bit 23 is set to 1, the subpage AP bits are disabled and the page tables have support for ARMv6 MMU features. Four new page table bits are added to support these features:

- The Not-Global (nG) bit, determines if the translation is marked as global (0), or process-specific (1) in the TLB. For process-specific translations the translation is inserted into the TLB using the current ASID, from the ContextID Register, CP15 c13.
- The Shared (S) bit, determines if the translation is for Non-Shared (0), or Shared (1) memory. This only applies to Normal memory regions. Device memory can be Shared or Non-Shared as determined by the TEX bits and the C and B bits.
- The Execute-Never (XN) bit, determines if the region is Executable (0) or Not-executable (1).

- Three access permission bits. The access permissions extension (APX) bit, provides an extra access permission bit.

All ARMv6 page table mappings support the TEX field.

ARMv6 page table format

Figure 5-6 shows the format of an ARMv6 first-level descriptor when subpages are enabled.

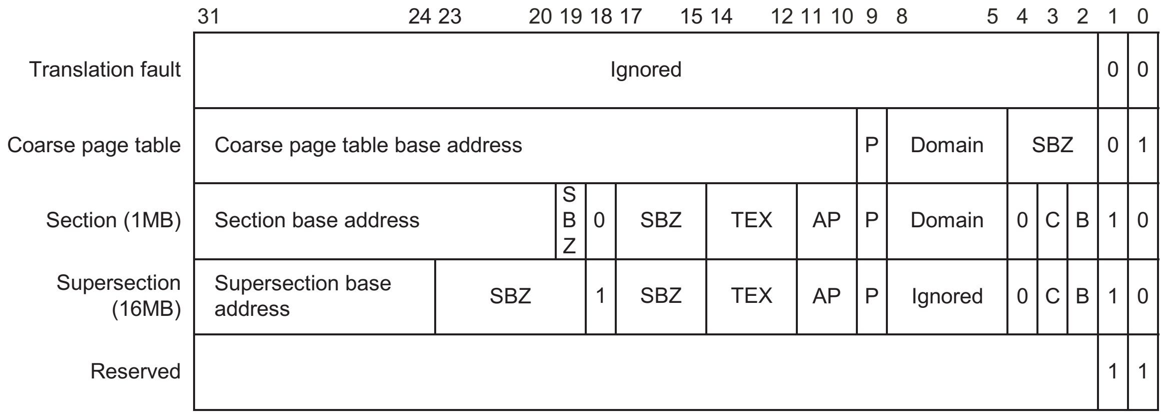

31 24 23 20 19 18 17 15 14 12 11 10 9 8 5 4 3 2 1 0
Translation fault
Ignored 0 0
Coarse page table
Coarse page table base address P Domain SBZ 0 1
Section (1MB)
Section base address SBZ 0 SBZ TEX AP P Domain 0 C B 1 0
Supersection (16MB)
Supersection base address SBZ 1 SBZ TEX AP P Ignored 0 C B 1 0
Reserved
1 1

Figure 5-6 ARMv6 first-level descriptor formats with subpages enabled

Figure 5-7 shows the format of an ARMv6 first-level descriptor when subpages are disabled.

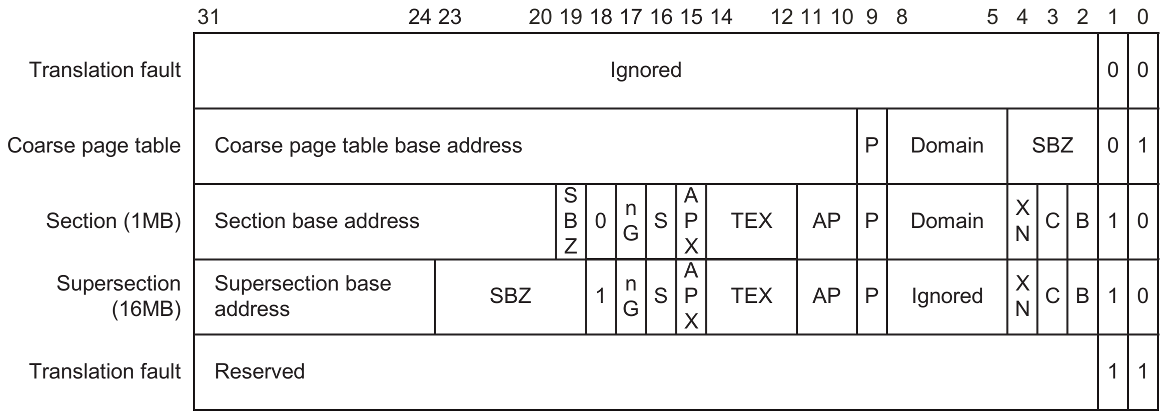

31 24 23 20 19 18 17 16 15 14 12 11 10 9 8 5 4 3 2 1 0
Translation fault
Ignored 0 0
Coarse page table
Coarse page table base address P Domain SBZ 0 1
Section (1MB)
Section base address SBZ 0 nG S APX TEX AP P Domain XN C B 1 0
Supersection (16MB)
Supersection base address SBZ 1 nG S APX TEX AP P Ignored XN C B 1 0
Translation fault
Reserved 1 1

Figure 5-7 ARMv6 first-level descriptor formats with subpages disabled

If the P bit is supported and set for the memory region, it indicates to the system memory controller that this memory region has ECC enabled.

In addition to the invalid translation, bits [1:0] = b00, translations for the reserved entry, bits [1:0] = b11, result in a translation fault.

Bit 18 of the first-level descriptor selects between a 1MB section and a 16MB supersection. For details of supersections see *Supersections* on page 5-6.

Figure 5-8 shows the format of an ARMv6 second-level descriptor.

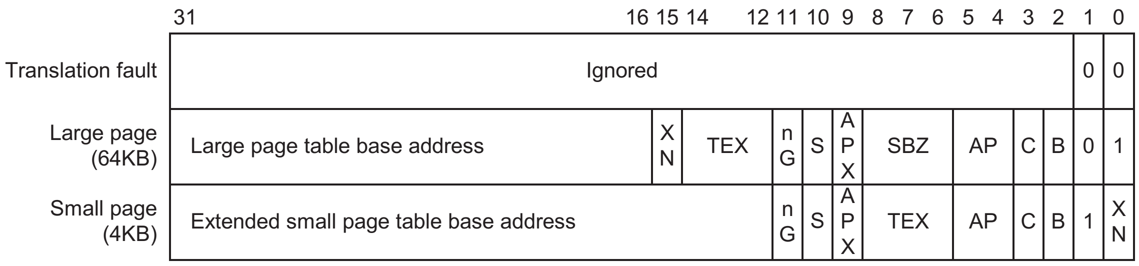


Figure 5-8 ARMv6 second-level descriptor format

Figure 5-9 on page 5-46 shows an overview of the section, supersection, and page translation process using ARMv6 descriptors.

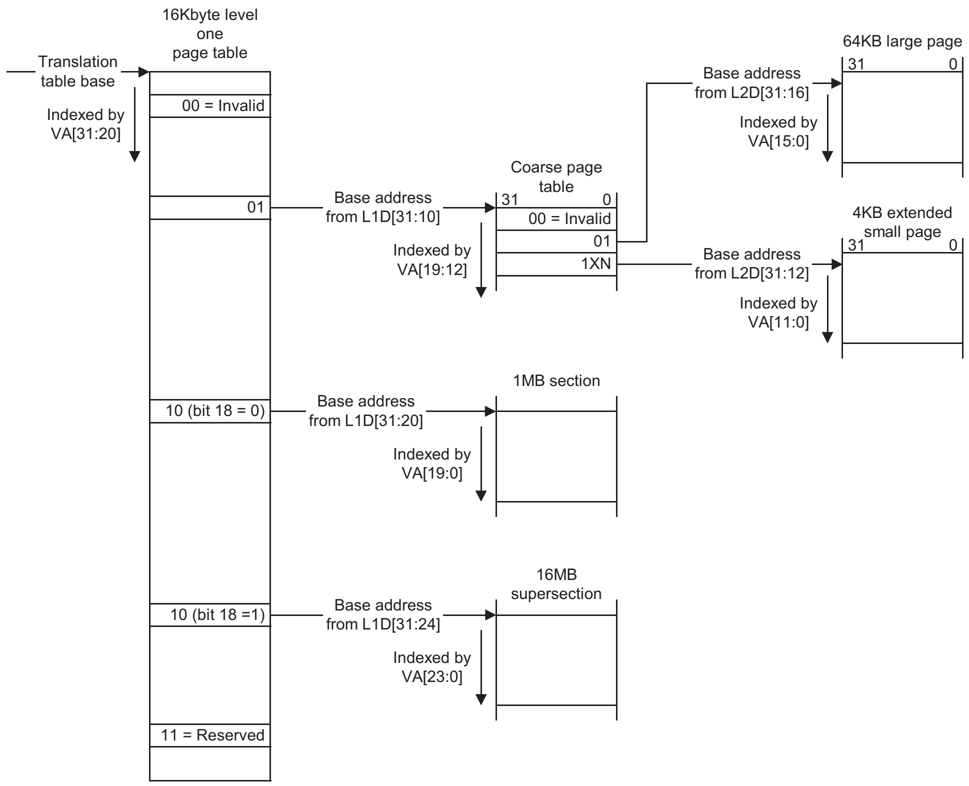


Figure 5-9 ARMv6 section, supersection, and page translation

5.11.3 Restrictions on page table mappings for the instruction cache

The instruction cache is virtually indexed and physically tagged to reduce the cache latency on nonsequential accesses. This implies that for 32KB and 64KB cache configurations, 4KB pages can cause aliasing problems.

To prevent this you must ensure the following restrictions are applied:

- If multiple Virtual Addresses are mapped onto the same physical address then for all mappings of bits [13:12] the Virtual Addresses must be equal and the same as bits [13:12] of the physical address. The same physical address can be mapped by TLB entries of different page sizes, including page sizes over 4KB.

- Alternatively, if all mappings to a physical address are of a page size equal to 4KB, then the restriction that bits [13:12] of the Virtual Address must equal bits [13:12] of the physical address is not necessary. Bits [13:12] of all Virtual Address aliases must still be equal.

There is no restriction on the more significant bits in the Virtual Address equalling those in the physical address.

If two different VA with different values for bits 12 and 13 point to the same PA, this leads to aliases of the same PA being cached in two different indexes. This is harmless for an instruction cache because no data corruption can occur.

If a VA is remapped from a first PA to a second PA with different values for bits 12 and 13, this prevents mapping a given PA to a unique index. This causes problems only for a CP15 PA operation. Such operation s do not exist on MP11 CPUs.

5.12 MMU descriptors

To support sections and pages, the MPCore MMU uses a two-level descriptor definition. The first-level descriptor indicates whether the access is to a section or to a page table. If the access is to a page table, the MPCore MMU determines the page table type and fetches a second-level descriptor.

5.12.1 First-level descriptor address

The *Translation Table Base Control Register* (TTBCR) selects between the two possible first-level descriptor addresses created by the two *Translation Table Base Registers* (TTBR0 and TTBR1) and the Virtual Address from the ARM11 MPCore processor. Figure 5-10 on page 5-49 shows the creation of a first-level descriptor address.

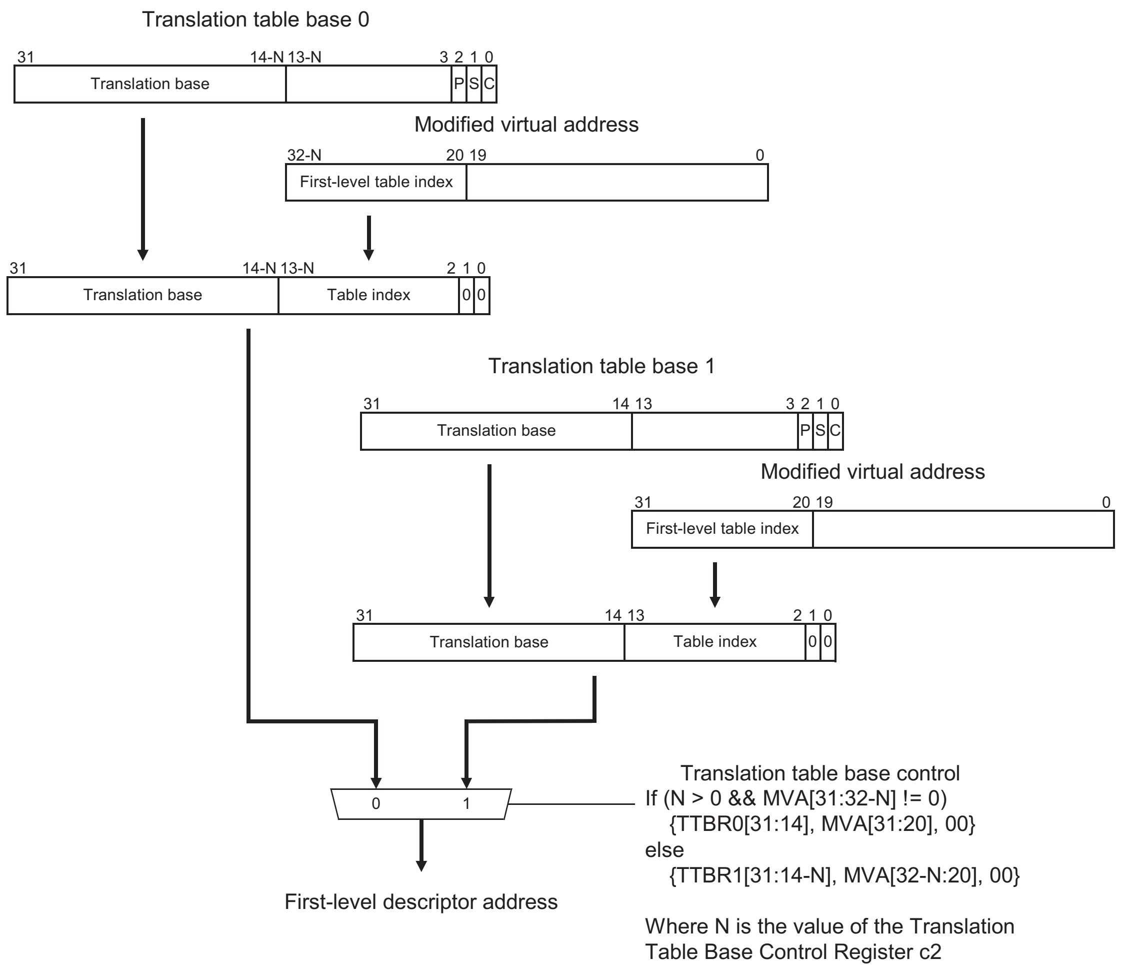


Figure 5-10 Creating a first-level descriptor address

5.12.2 First-level descriptor

Using the first-level descriptor address, a request is made to external memory. This returns the first-level descriptor. By examining bits [1:0] of the first-level descriptor, the access type is indicated as shown in Table 5-13.

Table 5-13 Access types from first-level descriptor bit values

Bit values	Access type
b00	Translation fault
b01	Page table base address
b10	Section base address
b11	Reserved, results in translation fault

First-level translation fault

If bits [1:0] of the first-level descriptor are b00 or b11, a translation fault is generated. This causes either a Prefetch Abort or Data Abort in the ARM11 MPCore processor. Prefetch Aborts occur in the instruction MMU. Data Aborts occur in the data MMU.

First-level page table address

If bits [1:0] of the first-level descriptor are b01, then a page table walk is required. This process is described in *Second-level page table walk* on page 5-52.

First-level section base address

If bits [1:0] of the first-level descriptor are b10, a request to a section memory block has occurred. Figure 5-11 on page 5-51 shows the translation process for a 1MB section using ARMv6 format (AP bits disabled).

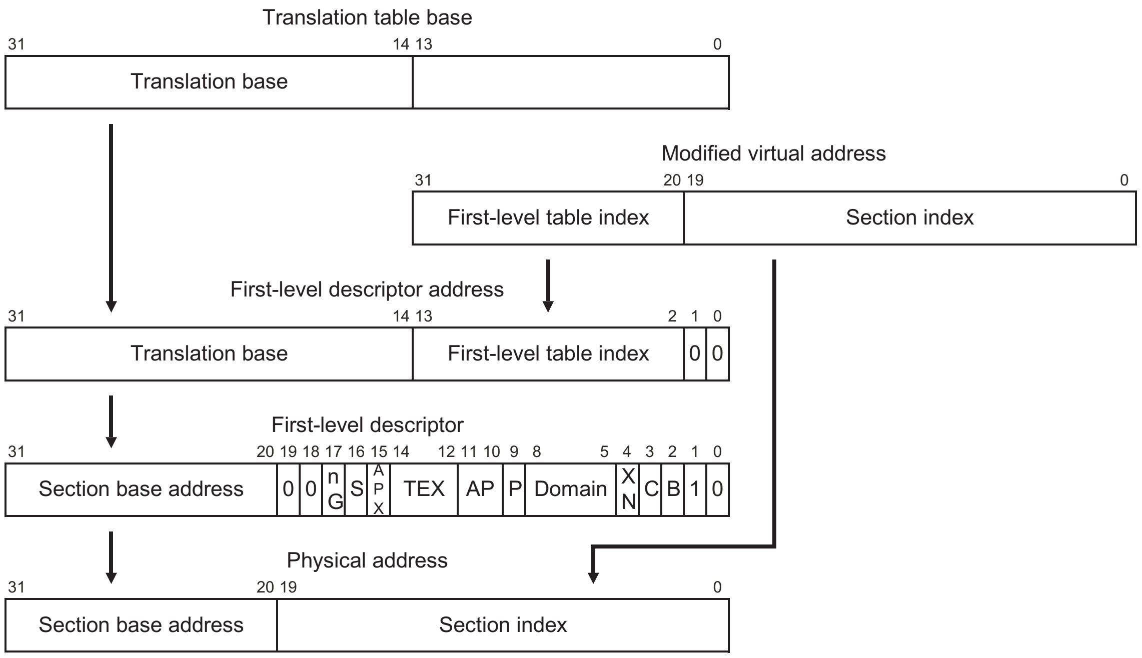


Figure 5-11 Translation for a 1MB section, ARMv6 format

Following the first-level descriptor translation, the physical address is used to transfer to and from external memory the data requested from and to the ARM11 MPCore processor. This is done only after the domain and access permission checks are performed on the first-level descriptor for the section. These checks are described in *Memory access control* on page 5-11.

Figure 5-12 on page 5-52 shows the translation process for a 1MB section using backwards-compatible format (AP bits enabled).

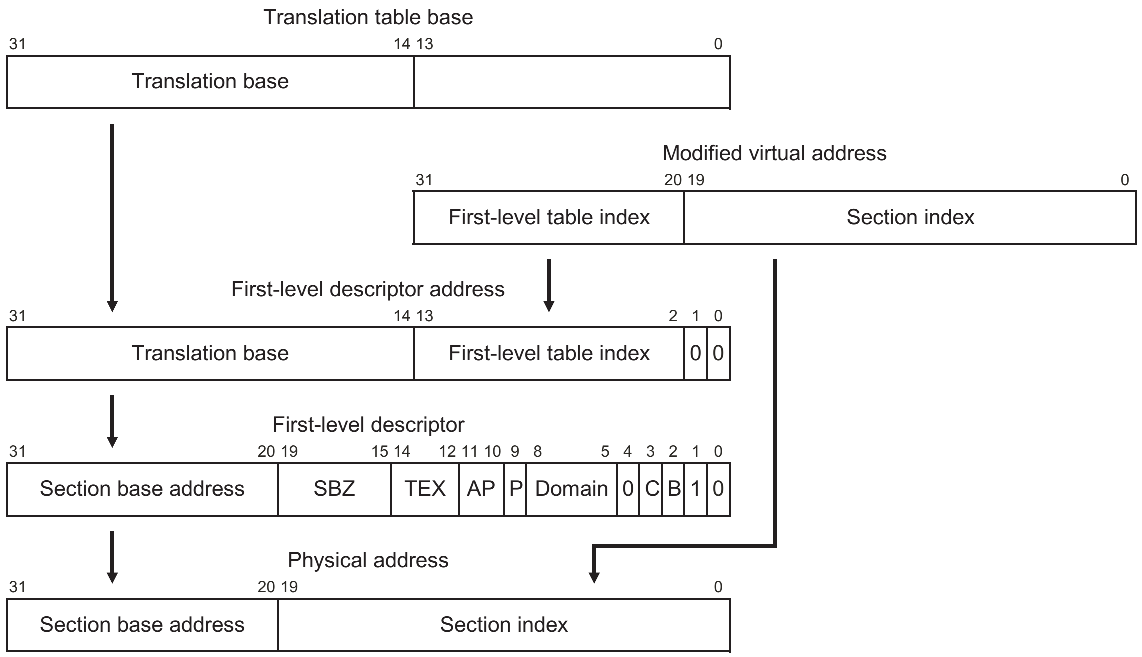


Figure 5-12 Translation for a 1MB section, backwards-compatible format

5.12.3 Second-level page table walk

If bits [1:0] of the first-level descriptor bits are b01, then a page table walk is required. The MMU requests the second-level page table descriptor from external memory. Figure 5-13 on page 5-53 shows how the second-level page table address is generated.

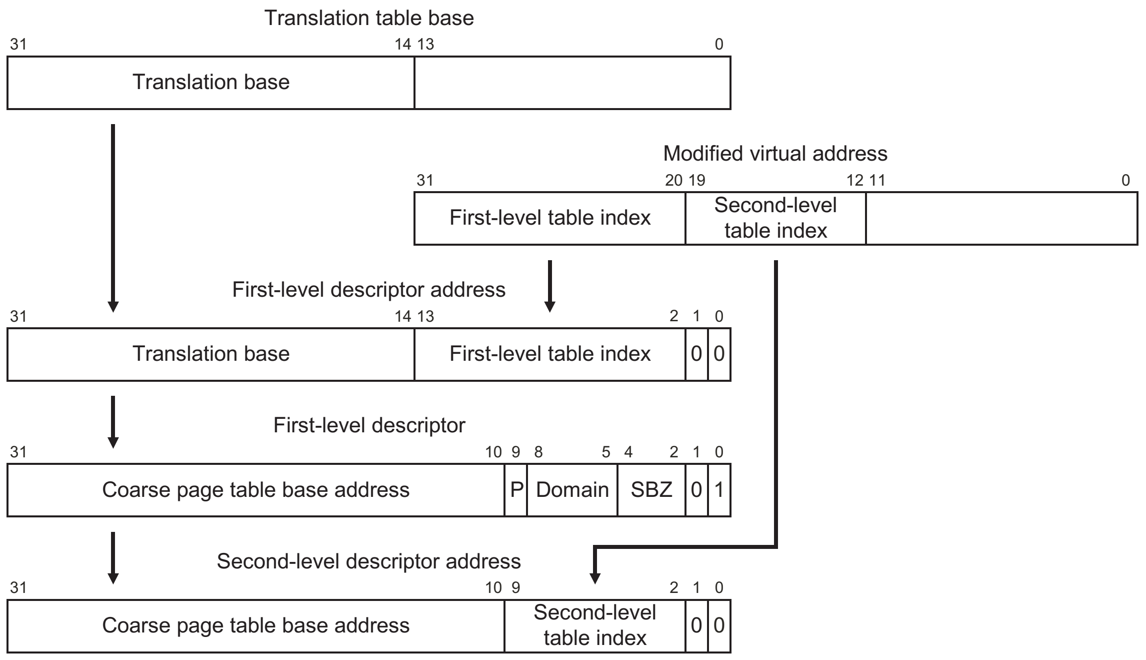


Figure 5-13 Generating a second-level page table address

When the page table address is generated, a request is made to external memory for the second-level descriptor.

By examining bits [1:0] of the second-level descriptor, the access type is indicated as shown in Table 5-14.

Table 5-14 Access types from second-level descriptor bit values

Descriptor format	Bit values	Access type
Both	b00	Translation fault
Backwards-compatible	b01	64KB large page
ARMv6	b01	64KB large page
Backwards-compatible	b10	4KB small page
ARMv6	b1XN	4KB extended small page
Backwards-compatible	b11	4KB extended small page

Second-level translation fault

If bits [1:0] of the second-level descriptor are b00, then a translation fault is generated. This generates an abort to the ARM11 MPCore processor, either a Prefetch Abort for the instruction side or a Data Abort for the data side.

Second-level large page base address

If bits [1:0] of the second-level descriptor are b01, then a large page table walk is required. Figure 5-14 shows the translation process for a 64KB large page using ARMv6 format (AP bits disabled).

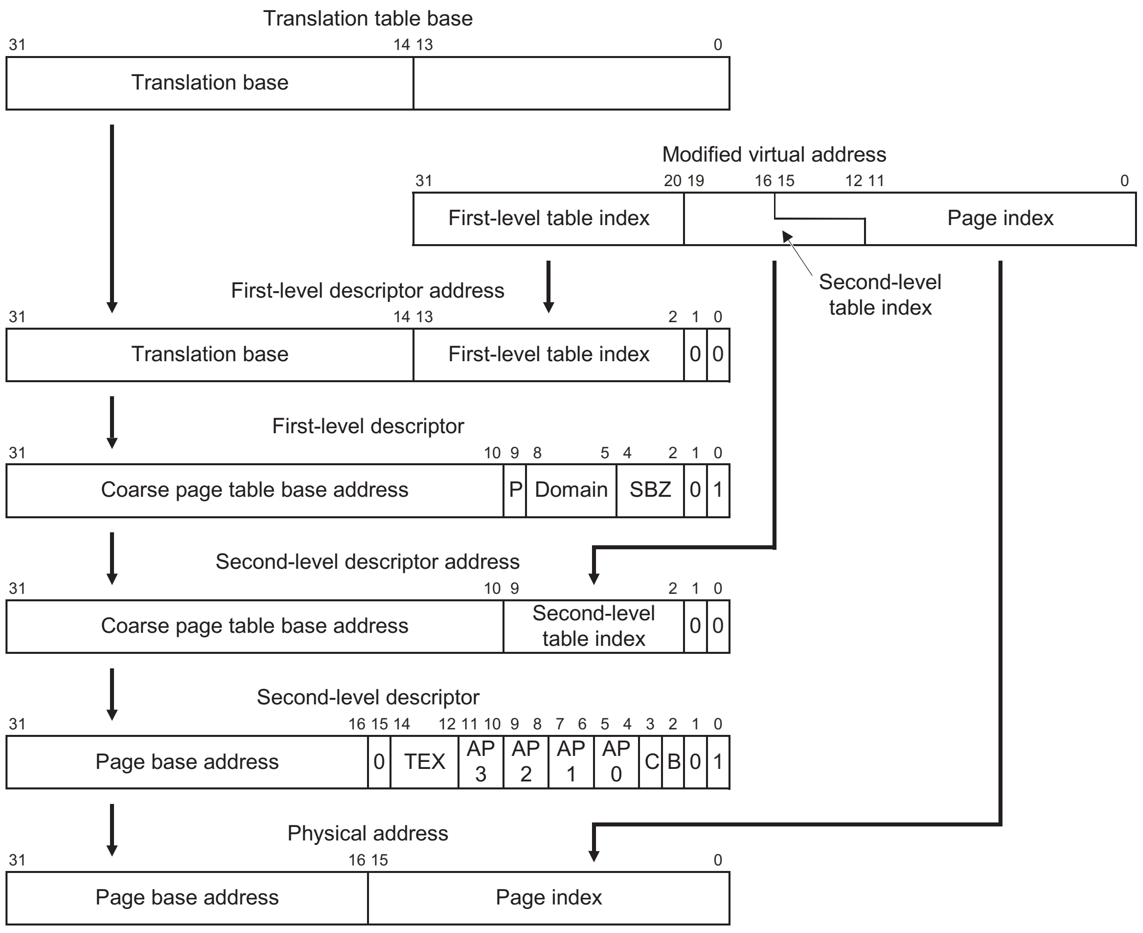


Figure 5-14 Large page table walk, ARMv6 format

Figure 5-15 on page 5-56 shows the translation process for a 64KB large page, or a 16KB large page subpage, using backwards-compatible format (AP bits enabled).

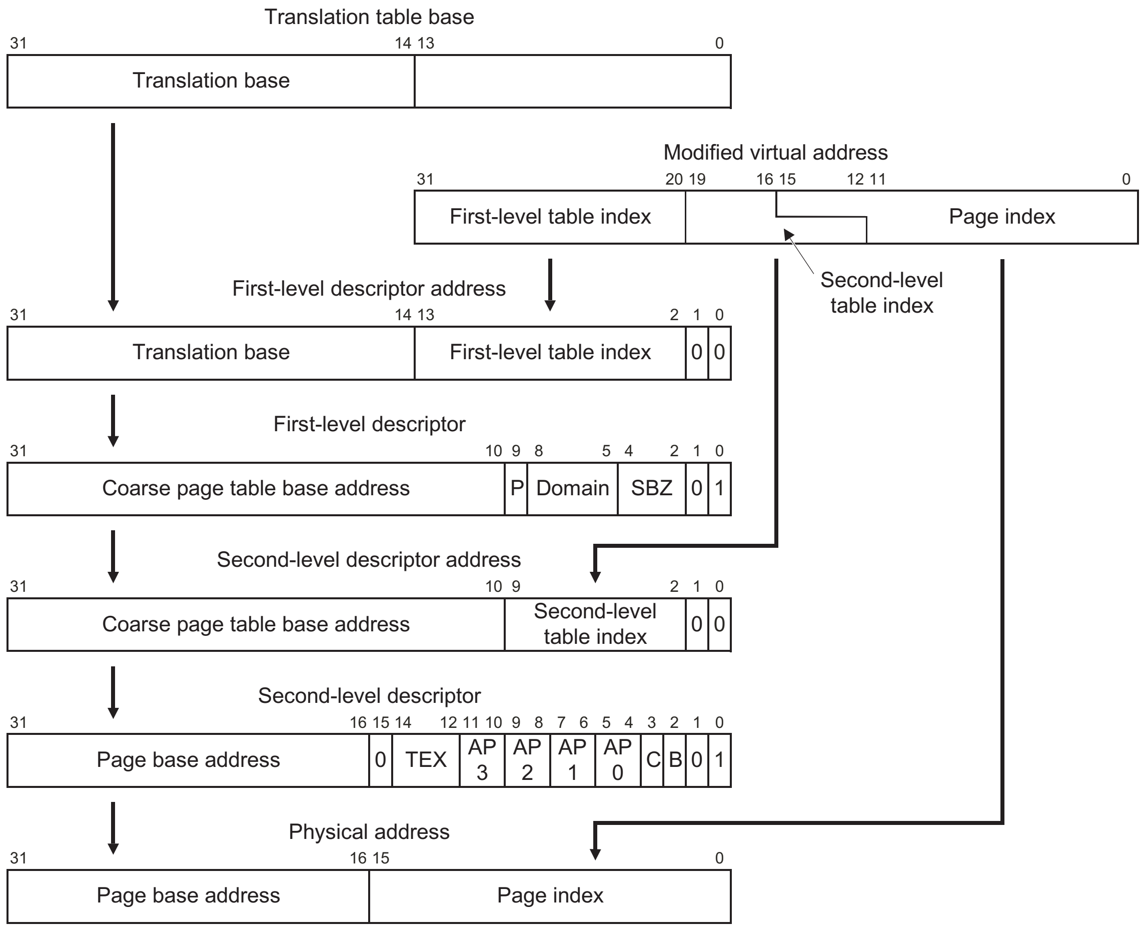


Figure 5-15 Large page table walk, backwards-compatible format

Using backwards-compatible format descriptors, the 64KB large page is generated by setting all of the AP bit pairs to the same values, AP3=AP2=AP1=AP0. If any one of the pairs are different, then the 64KB large page is converted into four 16KB large page subpages. The subpage access permission bits are chosen using the Virtual Address bits [15:14].

Second-level small page table walk

If bits [1:0] of the second-level descriptor are b10 for backwards-compatible format, then a small page table walk is required.

Figure 5-16 shows the translation process for a 4KB small page or a 1KB small page subpage using backwards-compatible format descriptors (AP bits enabled).

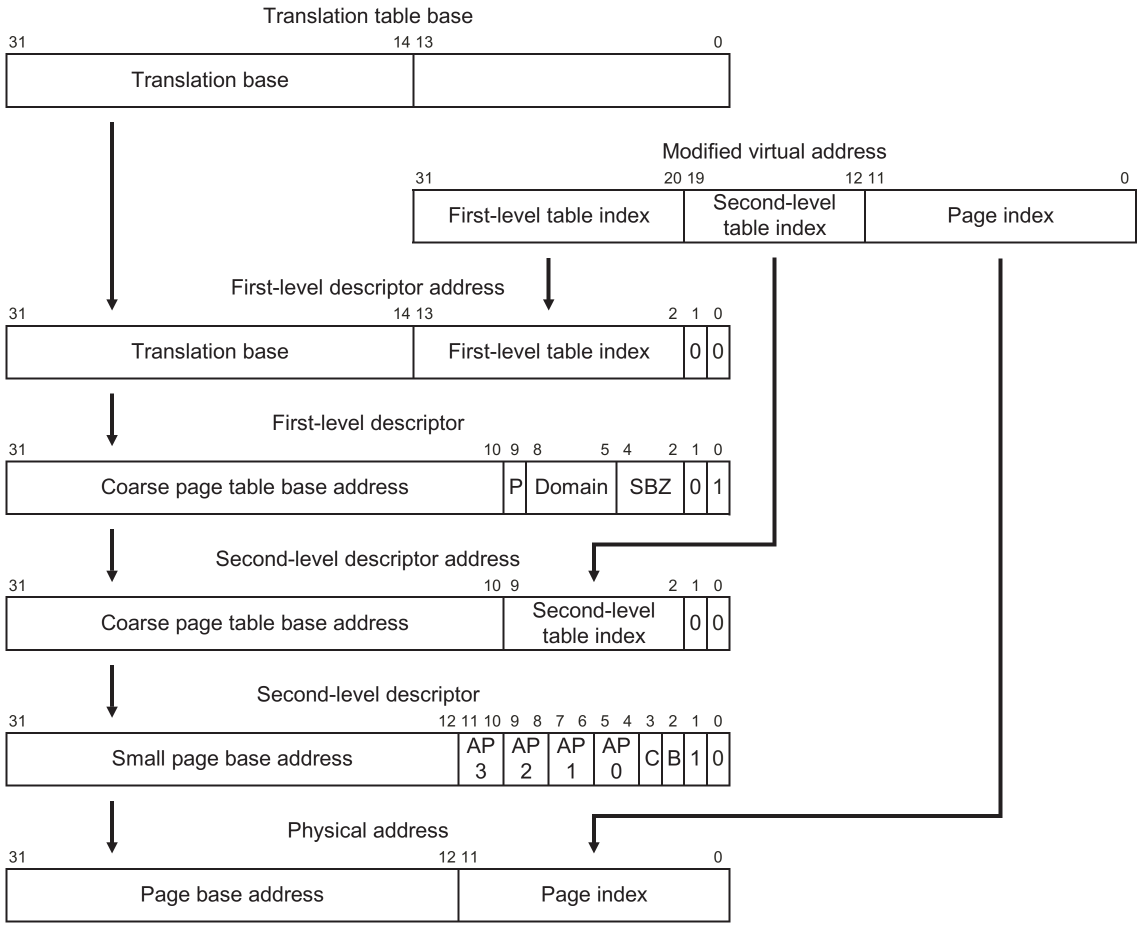


Figure 5-16 4KB small page or 1KB small subpage translations, backwards-compatible

Using backwards-compatible descriptors, the 4KB small page is generated by setting all of the AP bit pairs to the same values, AP3=AP2=AP1=AP0. If any one of the pairs are different, then the 4KB small page is converted into four 1KB small page subpages. The subpage access permission bits are chosen using the Virtual Address bits [11:10].

Second-level extended small page table walk

If bits [1:0] of the second-level descriptor are b1XN for ARMv6 format descriptors, or b11 for backwards-compatible descriptors, then an extended small page table walk is required. Figure 5-17 shows the translation process for a 4KB extended small page using ARMv6 format descriptors (AP bits disabled).

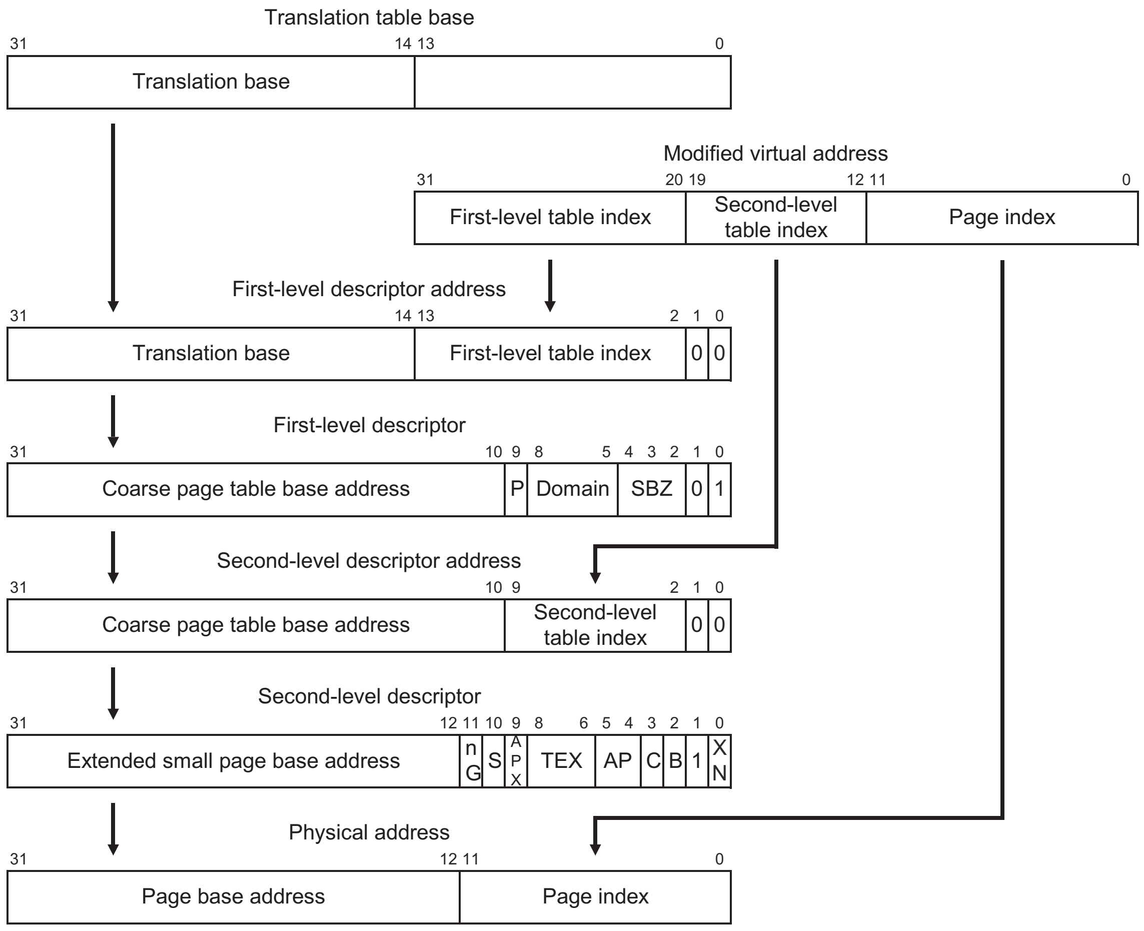


Figure 5-17 4KB extended small page translations, ARMv6 format

Figure 5-18 on page 5-59 shows the translation process for a 4KB extended small page or a 1KB extended small page subpage using backwards-compatible format descriptors (AP bits enabled).

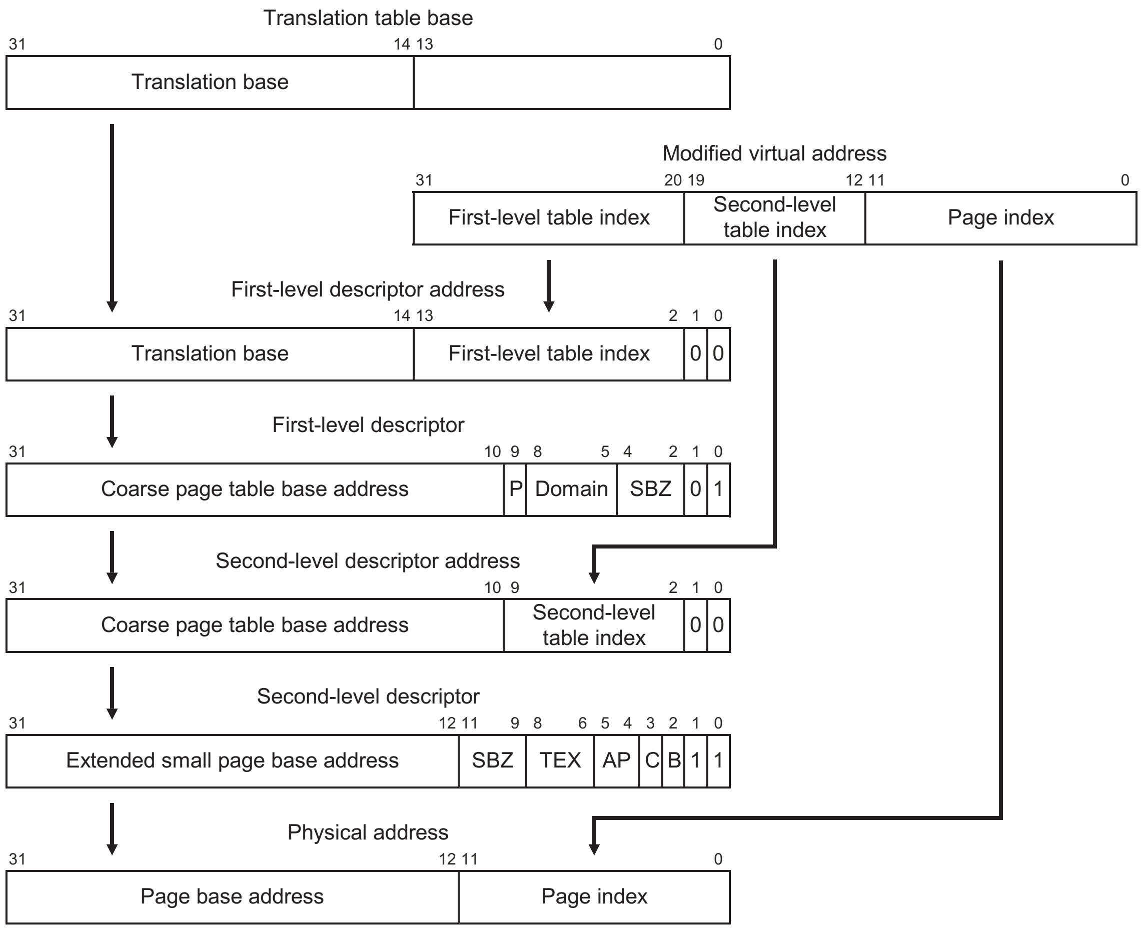


Figure 5-18 4KB extended small page or 1KB extended small subpage translations, backwards-compatible

Using backwards-compatible descriptors, the 4KB extended small page is generated by setting all of the AP bit pairs to the same values, AP3=AP2=AP1=AP0. If any one of the pairs are different, then the 4KB extended small page is converted into four 1KB extended small page subpages. The subpage access permission bits are chosen using the Virtual Address bits [11:10].

5.13 MMU software-accessible registers

The MMU is controlled by the system control coprocessor (CP15) registers, shown in Table 5-15, in conjunction with page table descriptors stored in memory.

You can access all the registers with instructions of the form:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

```
MCR p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

Where CRn is the system control coprocessor register. Unless specified otherwise, CRm and Opcode_2 Should Be Zero.

Table 5-15 CP15 register functions

Register	CRn	Bits	Description
TLB Type Register	c0	[23:16] ILsize, [15:8] DLsize, [0] U	The number of the TLB entries for the lockable TLB partitions is specified by the DLsize and ILsize fields respectively. See *c0, TLB Type Register* on page 3-14. U bit, unified or separate TLBs: 0 = unified TLB 1 = separate instruction and data TLBs.

Table 5-15 CP15 register functions (continued)

Register	CRn	Bits	Description
Control Register	c1	[0] M, [1] A, [3] W, [8] S, [9] R, [12]I [23] XP [28] TEX_REMAP [29] FORCE_AP	M bit, MMU enable/disable: 0 = MMU disabled 1 = MMU enabled. A bit, strict data address alignment fault enable/disable: 0 = Strict data address alignment fault checking disabled 1 = Strict data address alignment fault checking enabled. W bit, Write Buffer enable/disable. If implemented: 0 = Write Buffer disabled 1 = Write Buffer enabled. If not implemented, this bit reads as 1, writes ignored. S bit, system protection bit. Deprecated. See *c1, Control Register* on page 3-29. I bit, Instruction Cache enable/disable: 0 = Instruction Cache disabled 1 = Instruction Cache enabled. R bit, ROM protection. Deprecated. See *c1, Control Register* on page 3-29. XP bit, extended page table configuration: 0 = subpage AP bits enabled (backwards-compatible format descriptors used) 1 = subpage AP bits disabled, hardware translation tables support additional ARMv6 features (ARMv6 descriptors used). TEX_REMAP 0 = No remapping 1 = Remap registers are used for remapping FORCE_AP 0 = Access Bit not used 1 = AP[0] used as Access Bit.
Auxiliary Control Register	c0	[5] SMP bit	0 = AMP, processor is not part of coherency. 1 = SMP, processor is part of coherency.

Table 5-15 CP15 register functions (continued)

Register	CRn	Bits	Description
Translation Table Base Register 0	c2	[31:14-N] TTBR0	Pointer to the first-level translation table base 0 address for accessing page tables for process-specific addresses. N is the value of the Translation Table Base Control Register 2. It determines the boundary address of the translation table: If N = 0, the page table must reside on a 16KB boundary If N = 1, the page table must reside on a 8KB boundary ... If N = 7, the page table must reside on a 128-byte boundary. See *c2, Translation Table Base Register 0* on page 3-37.
Translation Table Base Register 1	c2	[31:14] TTBR1	Pointer to the first-level translation table base 0 address for accessing page tables for system and I/O addresses. See *c2, Translation Table Base Register 0* on page 3-37.
Translation Table Base Control Register	c2	[2:0] N	Translation table base register control: 0 = use TTBR0. Backwards compatible with ARMv5. 1 = if VA [31] = b0, use TTBR0, otherwise use TTBR1 2 = if VA[31:30] = b00, use TTBR0, otherwise use TTBR1 ... 7 = if VA[31:25] = b0000000, use TTBR0, otherwise use TTBR1. See *c2, Translation Table Base Control Register* on page 3-39.
Domain Access Control Register	c3	[31:0] D15-D0	Comprises 16 2-bit fields. Each field defines the access control attributes for one of 16 domains, D15–D0. See *c3, Domain Access Control Register* on page 3-41.
Data Fault Status Register (DFSR)	c5	[12]SD [11]Not read/write [10]Status [7:4] Domain, [3:0] Status	Indicates the cause of a Data Abort and the domain number of the aborted access, when a Data Abort occurs. Bits [7:4] specify which of the 16 domains (D15–D0) was being accessed when a fault occurred. Bits[12:10] and bits [3:0] indicate the type of access being attempted. The value of all other bits is Unpredictable. The encoding of these bits is shown in *c5, Data Fault Status Register* on page 3-42.
Instruction Fault Status Register (IFSR)	c5	[12]SD [10]Status [3:0] Status	Bit[12] and bit [10], and bits [3:0] indicate the type of access being attempted. The value of all other bits is Unpredictable. The encoding of these bits is shown in *c5, Instruction Fault Status Register* on page 3-44.

Table 5-15 CP15 register functions (continued)

Register	CRn	Bits	Description
Fault Address Register (FAR)	c6	[31:0] Data fault address	Holds the Modified Virtual Address associated with the access that caused the data fault. See *MMU fault checking* on page 5-33 for instructions to address the FAR. See *c6, Fault Address Register* on page 3-45 for details of the address stored for each type of fault.
Watchpoint Fault Address Register (WFAR)	c6	[31:0] Watchpoint fault address	Holds the Virtual Address of the instruction that triggered the watchpoint. See *c6, Watchpoint Fault Address Register* on page 3-45 for details.
Cache Operations Register	c7	[31:5] MVA or [31:30] Index, [S+2:3] Set Where S is $\log_2$ of the Size Field in the *c0, TLB Type Register* on page 3-14. [0] selects TCM when set to 1 or cache when set to 0.	A write-only register that you can use to control a memory barrier, Data Cache, and Write Buffer operations. Also used to control operations on prefetch buffers, and branch target caches, if they are implemented. Instructions to this register are in one of two formats: • MVA format • Index/Set format. See *c7, Cache Operations Register* on page 3-46 for details.
TLB Operations Register	c8	[31:10] MVA [7:0] ASID	Writing to this register causes the MMU to perform TLB maintenance operations. Three functions are provided, selected by the value of the Opcode_2 field: b000 = invalidate all the (unpreserved) entries in a TLB b001 = invalidate a specific entry b010 = invalidate entry on ASID match. Reading from this register is Unpredictable. See *c8, TLB Operations Register* on page 3-54.
TLB Lockdown Register	c10	[31:29] SBZ [28:26] Victim [0] P	The Victim field specifies which TLB entry in the lockdown region is replaced by the translation table walk result generated by the next TLB miss. Any translation table walk results written to TLB entries while P = 1 are protected from being invalidated by r8 Invalidate TLB operations. Translation table walk results written to TLB entries while P = 0 are invalidated normally by r8 Invalidate TLB operations.
TLB remap registers	c10	-	The Primary Remap Register and the Normal Remap Register are used to remap page table attributes set by the page table descriptors. See *c10, Memory remap registers* on page 3-60.

Table 5-15 CP15 register functions (continued)

Register	CRn	Bits	Description
FCSE PID Register	c13	[31:25] FCSE PID	This register controls the fast context switch extension. See *c13, FCSE PID Register* on page 3-63.
ContextID Register	c13	[31:8] ProcID [7:0] ASID	The bottom eight bits of this register contain the ASID of the currently running process. The ProcID bits extend the ASID. See *c13, Context ID Register* on page 3-66.

Note

All the CP15 MMU registers, except CP15 c7 and CP15 c8, contain state that you read using MRC instructions and written to using MCR instructions. Registers c5 and c6 are also written by the MMU. Reading CP15 c7 and c8 is Unpredictable.

5.14 MMU and write buffer

During any translation table walk the MMU has access to external memory. Before the table walk occurs, the write buffer must be flushed of any related writes related to the page descriptors to avoid read-after-write hazards.

When either the instruction MMU or data MMU contains valid TLB entries that are being modified, those TLB entries must be invalidated by software, and the Write Buffer drained using the Data Synchronization Barrier instruction before the new section or page is accessed.

Chapter 6
Program Flow Prediction

This chapter outlines how program flow prediction locates branches in the instruction stream and the strategies used for determining if a branch is likely to be taken or not. It also describes the two architecturally-defined SWI functions required for backwards-compatibility with earlier architectures for flushing the *Prefetch Unit* (PU) buffers. It contains the following sections:

6.1 About program flow prediction

Program flow prediction in ARM11 MPCore processors is carried out by:

The core	Implements static branch prediction and the Return Stack.
The Prefetch Unit	Implements dynamic branch prediction.

The ARM11 MPCore processor is responsible for handling branches the first time they are executed, that is, when no historical information is available for dynamic prediction by the PU.

The core makes static predictions about the likely outcome of a branch early in its pipeline and then resolves those predictions when the outcome of conditional execution is known. Condition codes are evaluated at three points in the core pipeline, and branches are resolved as soon as the flags are guaranteed not to be modified by a preceding instruction.

When a branch is resolved, the core passes information to the PU so that it can make a *Branch Target Address Cache* (BTAC) allocation or update an existing entry as appropriate. The core is also responsible for identifying likely procedure calls and returns to predict the returns. It can handle nested procedures up to three deep.

The core includes:

- a *Static Branch Predictor* (SBP)
- a *Return Stack* (RS)
- branch resolution logic
- a BTAC update interface to the PU.

The MPCore PU is responsible for fetching instructions from the memory system as required by the integer unit, and coprocessors. The PU buffers up to three instructions in its FIFO to:

- detect branch instructions ahead of the integer unit requirement
- dynamically predict those that it considers are to be taken
- provide branch folding of predicted branches if possible.

This reduces the cycle time of the branch instructions, so increasing processor performance.

The PU includes:

- a BTAC
- branch update and allocate logic
- a *Dynamic Branch Predictor* (DBP), and associated update mechanism
- branch folding logic.

It is responsible for providing the core with instructions, and for requesting cache accesses. The pattern of cache accesses is based on the predicted instruction stream as determined by the dynamic branch prediction mechanism or the core flush mechanism.

The BTAC can:

- be globally flushed by a CP15 instruction
- have individual entries flushed by a CP15 instruction
- be enabled or disabled by a CP15 instruction.

For details of CP15 instructions see Chapter 3 *Control Coprocessor CP15*.

The PU also handles the cache access multiplexing for CP15 instruction handling

The PU prefetches all instruction types regardless of the state of the core. That is, for ARM state, Thumb state, or Java state. However the rate of draining of the PU is a function of these states, and the functioning of the branch prediction hardware is a function of the state. The branch prediction is performed in all states, ARM, Thumb, and Java but branch folding operates only in ARM state.

The PU is responsible for fetching the instruction stream as dictated by:

- the Program Counter
- the dynamic branch predictor
- static prediction results in the core
- procedure calls and returns signaled by the Return Stack residing in the core
- exceptions, instruction aborts, and interrupts signaled by the core.

6.2 Branch prediction

In ARM processors that have no PU, the target of a branch is not known until the end of the Execute stage. At the Execute stage it is known whether or not the branch is taken. The best performance is obtained by predicting all branches as not taken and filling the pipeline with the instructions that follow the branch in the current sequential path. In ARM processors without a PU, an untaken branch requires one cycle and a taken branch requires three or more cycles.

Branch prediction enables the detection of branch instructions before they enter the integer unit. This permits the use of a branch prediction scheme that closely models actual conditional branch behavior.

The increased pipeline length of the ARM11 MPCore processor makes the performance penalty of any changes in program flow, such as branches or other updates to the PC, more significant than was the case on the ARM9TDMI or ARM1020T cores. Therefore, a significant amount of hardware is dedicated to prediction of these changes. Two major classes of program flow are addressed in the MPCore prediction scheme:

1. Branches (including BL, and BLX immediate), where the target address is a fixed offset from the program counter. The prediction amounts to an examination of the probability that a branch passes its condition codes. These branches are handled in the Branch Predictors.

2. Loads, Moves, and ALU operations writing to the PC, which can be identified as being likely to be a return from a procedure call. Two identifiable cases are Loads to the PC from an address derived from r13 (the stack pointer), and Moves or ALU operations to the PC derived from r14 (the Link Register). In these cases, if the calling operation can also be identified, the likely return address can be stored in a hardware implemented stack, termed a *Return Stack* (RS). Typical calling operations are BL and BLX instructions. In addition Moves or ALU operations to the Link Register from the PC are often preludes to a branch that serves as a calling operation. The Link Register value derived is the value required for the RS. This was most commonly done on ARMv4T, before the BLX <register> instruction was introduced in ARMv5T.

Branch prediction is required in the design to reduce the core CPI loss that arises from the longer pipeline. To improve the branch prediction accuracy, a combination of static and dynamic techniques is employed. It is possible to disable the predictors.

6.2.1 Enabling program flow prediction

The enabling of program flow prediction is controlled by the CP15 Register c1 Z bit (bit 11), which is set to 0 on Reset. See *c1, Control Register* on page 3-29. The return stack, dynamic predictor, and static predictor can also be individually controlled using the Auxiliary Control Register. See *c1, Auxiliary Control Register* on page 3-33.

6.2.2 Dynamic branch predictor

The first line of branch prediction in the ARM11 MPCore processor is dynamic, through a simple BTAC. It is virtually addressed and holds virtual target addresses. In addition, a two bit value holds the predicted direction of the branch. If the address mappings change, this cache must be flushed. A dynamic branch predictor flush is included in the CP15 coprocessor control instructions.

A BTAC works by storing the existence of branches at particular locations in memory. The branch target address and a prediction of whether or not it might be taken is also stored.

The BTAC provides dynamic prediction of branches, including BL and BLX instructions in both ARM, Thumb, and Java states. The BTAC is a 128-entry direct-mapped cache structure used for allocation of Branch Target Addresses for resolved branches. The BTAC uses a 2-bit saturating prediction history scheme to provide the dynamic branch prediction. When a branch has been allocated into the BTAC, it is only evicted in the case of a capacity clash. That is, by another branch at the same index.

The prediction is based on the previous behavior of this branch. The four possible states of the prediction bits are:

- strongly predict branch taken
- weakly predict branch taken
- weakly predict branch not taken
- strongly predict branch not taken.

The history is updated for each occurrence of the branch. This updating is scheduled by the core when the branch has been resolved.

Branch entries are allocated into the BTAC after having been resolved at Execute. BTAC hits enable branch prediction with zero cycle delay. When a BTAC hit occurs, the Branch Target Address stored in the BTAC is used as the Program Counter for the next Fetch. Both branches resolved taken and not taken are allocated into the BTAC. This enables the BTAC to do the most useful amount of work and improves performance for tight backward branching loops.

6.2.3 Static branch predictor

The second level of branch prediction in the ARM11 MPCore processor uses static branch prediction that is based solely on the characteristics of a branch instruction. It does not make use of any history information. The scheme used in the ARM11 MPCore processor predicts that all forward conditional branches are not taken and all backward branches are taken. Around 65% of all branches are preceded by enough non-branch cycles to be completely predicted.

Branch prediction is performed only when the Z bit in CP15 Register c1 is set to 1. See *c1, Control Register* on page 3-29 for details of this register. Dynamic prediction works on the basis of caching the previously seen branches in the BTAC, and like all caches suffers from the compulsory miss that exists on the first encountering of the branch by the predictor. A second, static predictor is added to the design to counter these misses, and to mop-up any capacity and conflict misses in the BTAC. The static predictor amounts to an early evaluation of branches in the pipeline, combined with a predictor based on the direction of the branches to handle the evaluation of condition codes that are not known at the time of the handling of these branches. Only items that have not been predicted in the dynamic predictor are handled by the static predictor.

The static branch predictor is hard-wired with backward branches being predicted as taken, and forward branches as not taken. The SBP looks at the MSB of the branch offset to determine the branch direction. Statically predicted taken branches incur a one-cycle delay before the target instructions start refilling the pipeline. The SBP works in both ARM and Thumb states. The SBP does not function in Java state. It can be disabled using CP15 Register c1. See *c1, Control Register* on page 3-29.

6.2.4 Branch folding

Branch folding is a technique where, on the prediction of most branches, the branch instruction is completely removed from the instruction stream presented to the execution pipeline. Branch folding can significantly improve the performance of branches, taking the CPI for branches significantly below 1.

When the PU prefetches a predicted taken instruction, it evaluates if the branch:

- is not a BL or BLX instruction (to avoid losing the link)
- does not point to a code sequence that contains a branch in the first two instructions
- is not breakpointed
- is not aborted.

If all of the conditions are met, the branch is folded on the subsequent prefetches of the branch.

6.2.5 Incorrect predictions and correction

Branches are resolved at or before the Ex3 stage of the core pipeline. A misprediction causes the pipeline to be flushed, and the correct instruction stream to be fetched. If branch folding is implemented, the failure of the condition codes of a folded branch causes the instruction that follows the folded branch to fail. Whenever a potentially incorrect prediction is made, the following information, necessary for recovering from the error, is stored:

- a fall-through address in the case of a predicted taken branch instruction
- the branch target address in the case of a predicted not taken branch instruction.

The PU passes the conditional part of any folded branch into the integer unit. This enables the integer unit to compare these bits with the processor flags and determine if the prediction was correct or not. If the prediction was incorrect, the integer unit flushes the PU and requests that prefetching begins from the stored recovery address.

6.3 Return stack

A return stack is used for predicting the class of program flow changes that includes loads, moves, and ALU operations, writing to the PC that can be identified as being likely to be a procedure call or return.

The return stack is a three-entry circular buffer used for the prediction of procedure calls and procedure returns. Only unconditional procedure returns are predicted.

When a procedure call instruction is predicted, the return address is taken from the Execute stage of the pipeline and pushed onto the return stack. The instructions recognized as procedure calls are:

- `BL <dest>`
- `BLX <dest>`
- `BLX <reg>`.

The first two instructions are predicted by the BTAC, unless they result in a BTAC miss. The third instruction is not predicted. The SBP predicts unconditional procedure calls as taken, and conditional procedure calls as not taken.

When a procedure return instruction is predicted, an instruction fetch from the location at the top of the return stack occurs, and the return stack is popped. The instructions recognized as procedure returns are:

- `BX r14`
- `LDM sp!, {...,pc}`
- `LDR pc, [sp...]`.

The SBP only predicts procedure returns that are always predicted as taken.

Two classes of return stack mispredictions can exist:

- condition code failures of the return operation
- incorrect return location.

In addition, an empty return stack gives no prediction.

6.4 Instruction Memory Barrier (IMB) instruction

In some circumstances it is likely that the prefetch unit pipeline and the core pipeline contain out-of-date instructions. In these circumstances the prefetch unit must be flushed. The *Instruction Memory Barrier* (IMB) instruction provides a way to do this for the ARM1020T processor. The ARM11 MPCore processor maintains this capability for backwards compatibility with the ARM1020T.

To implement the two IMB instructions, you must include processor-specific code in the SWI handler:

`IMB`	The IMB instruction flushes all information about all instructions.
`IMBRange`	When only a small area of code is altered before being executed the IMBRange instruction can be used to efficiently and quickly flush any stored instruction information from addresses within a small range. By flushing only the required address range information, the rest of the information remains to provide improved system performance.

These instructions are implemented as calls to specific SWI numbers:

`IMB`	`SWI 0xF00000`
`IMBRange`	`SWI 0xF00001`.

6.4.1 Generic IMB use

Use SWI functions to provide a well-defined interface between code that is:

- independent of the ARM processor implementation it is running on
- specific to the ARM processor implementation it is running on.

The implementation-independent code is provided with a function that is available on all processor implementations using the SWI interface, and that can be accessed by privileged and, where appropriate, non-privileged (User mode) code.

Using SWIs to implement the IMB instructions means that any code that is written now is compatible with any future processors, even if those processors implement IMB in different ways. This is achieved by changing the operating system SWI service routines for each of the IMB SWI numbers that differ from processor to processor.

6.5 ARM1020T or later IMB implementation

For ARM1020T or later processors, executing the SWI instruction is sufficient in itself to cause IMB operation. Also, for ARM1020T or later, both the IMB and the IMBRange instructions flush all stored information about the instruction stream.

This means that all IMB instructions can be implemented in the operating system by returning from the IMB or IMBRange service routine and that the service routines can be exactly the same. The following service routine code can be used:

```
IMB_SWI_handler
IMBRange_SWI_handler
MOVS    PC, R14_svc  ; Return to the code after the SWI call
```

Note

- In new code, you are strongly encouraged to use the IMBRange instruction whenever the changed area of code is small, even if there is no distinction between it and the IMB instruction on ARM1020T or ARM11 MPCore processors. Future processors might implement the IMBRange instruction in a much more efficient and faster manner, and code migrated from the ARM920T core is likely to benefit when executed on these processors.
- ARM11 MPCore processors implement a Flush Prefetch Buffer operation that is user-accessible and acts as an IMB. For more details see *c7, Cache Operations Register* on page 3-46.

6.5.1 Execution of IMB instructions

This section comprises three examples that show what can happen during the execution of IMB instructions. The pseudo code in the square brackets shows what happens to execute the IMB instruction (or IMBRange) in the SWI handler.

Example 6-1 shows how code that loads a program from a disk, and then branches to the entry point of that program, must execute an IMB instruction between loading the program and trying to execute it.

Example 6-1 Loading code from disk

```
IMBEQU 0xF00000
.
.
; code that loads program from disk
.
    .
    SWI MB
        [branch to IMB service routine]
        [perform processor-specific operations to execute IMB]
        [return to code]
        .
    MOV PC, entry_point_of_loaded_program
        .
        .
```

Compiled BitBlt routines optimize large copy operations by constructing and executing a copying loop that has been optimized for the exact operation wanted. When writing such a routine an IMB is required between the code that constructs the loop and the actual execution of the constructed loop. Example 6-2 shows this.

Example 6-2 Running BitBlt code

```
IMBRange EQU 0xF00001.
.
; code that constructs loop code
; load R0 with the start address of the constructed loop
; load R1 with the end address of the constructed loop
SWI MBRange
    [branch to IMBRange service routine]
    [read registers R0 and R1 to set up address range parameters]
    [perform processor-specific operations to execute IMBRange]
    [within address range]
    [return to code]
; start of loop code
.
.
```

When writing a self-decompressing program, an IMB must be issued after the routine that decompresses the bulk of the code and before the decompressed code starts to be executed. Example 6-3 shows this.

Example 6-3 Self-decompressing code

```
IMBEQU0xF00000
.
.
; copy and decompress bulk of code
SWI IMB
; start of decompressed code
.
```

.
.

Chapter 7
Level 1 Memory System

This chapter describes the Level 1 Memory System. It contains the following sections:

7.1 Coherency protocol

The coherency protocol is based on a MESI-type protocol. MESI is a write-invalidate cache protocol. When writing to a shared location, the related coherent cache line is invalidated in all other caches in the L1 memory system. Exclusive use is lost when another processor tries to read that shared location.

Coherent cache line attributes can be:

Modified The current version of the coherent cache line resides within this cache, and no other copies of the same memory location exist. Its contents are not up to date with main memory.

Exclusive Signifies to the cache controller that the main memory location is up to date with the contents of the cache and that no other copies of the same memory location exist.

Shared This coherent cache line is present in the cache and up to date with main memory.

Invalid This coherent cache line is not present in the cache.

7.1.1 Optimizations

The ARM11 MPCore processor uses the following optimizations of the MESI-based coherency protocol:

Direct Data Intervention

DDI enables copying clean data from one MP11 CPU L1 data cache to another MP11 CPU without accessing external memory. This reduces read after read activity from the Level 1 cache to the Level 2 cache.

Duplicated tag RAMs

Duplicated tag RAMs are duplicated versions of L1 tag RAMs used by the SCU to check for data availability before sending coherency commands to the relevant MP11 CPUs. Coherency commands are sent only to MP11 CPUs that must update their coherent data cache.

Migratory lines

The migratory lines feature enables moving dirty data from one MP11 CPU to another without writing to L2 and reading the data back in from external memory.

7.2 About the Level 1 data side memory system

Figure 7-1 shows the level 1 data side memory system block diagram.

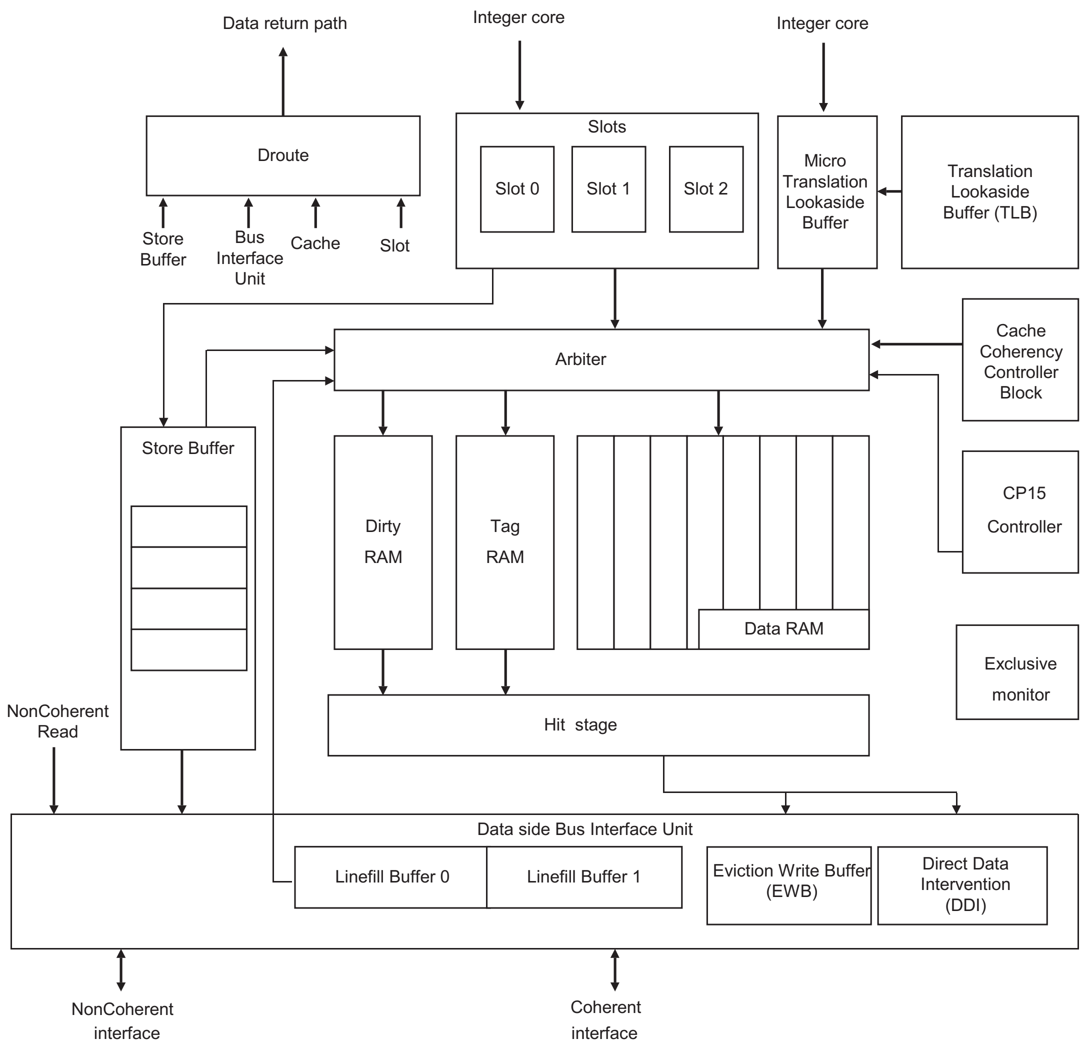


Figure 7-1 Level 1 data side memory system block diagram

The L1 data cache is organized as a physically indexed and physically tagged cache. The micro-TLB produces the Physical Address from the Virtual Address before performing the cache access.

This removes the need to use a software page coloring scheme on the data side when programming the MMU. See *Restrictions on page table mappings for the instruction cache* on page 5-46.

The data cache has been architected to ensure that all evictions and allocations are performed in one clock cycle, that is, the interface for these operations is 256 bits wide.

7.2.1 Slots Unit

The Slots Unit consists of three independent slots that hold the information of each outstanding memory request being performed by the integer core.

When a memory request (read or write access) is presented to the slots, it always goes through the following steps:

- The micro-TLB unit converts the Virtual Address to a Physical Address and gets cache and protection attributes. See *Micro TLB* on page 7-6.
- Using the cache and protection attributes, the Slots Unit then produces an MMU or alignment fault if needed.

After these steps, the read and write paths differ:

For a read memory request

The Slots Unit requests an access to the Data RAM and asks one Line Fill Buffer (LFB) for a line fill in case of a read miss.

For a write memory request

The Slots Unit sends the cache attributes, the burst information and the data for the access to the Store Buffer (STB). If the STB is not ready to take the write request, the memory transfer waits in the Slot Unit until the Store Buffer is ready to take it.

Cachable bursts are cut on cache line boundaries and noncachable bursts on 1KB boundaries. The Slots Unit computes the sequentiality information associated with the access. This information is used to avoid unnecessary tag RAM lookups for read bursts.

As an example, for a READ burst:

- For the first (nonsequential) access, all tag RAMs are enabled for the lookup. The Data RAM read is performed in parallel on all banks.
- For the first sequential access, all Data RAMs are also enabled because the way hit information is not yet available.
- For any further sequential accesses, only the data RAM containing the requested data is enabled based on the transfer size (8, 16, 32, or 64 bits).

7.2.2 Noncachable accesses and locked accesses

To avoid hazards and ordering issues some safety checks are done before starting noncachable or locked memory accesses.

Before starting any Strongly Ordered accesses, the Slots Unit ensures that:

- The Store Buffer is empty
- The Line Fill Buffers and Eviction Buffer are empty
- No other memory request is currently pending in the Slots Unit.

All Strongly Ordered memory accesses use the precise abort mechanism.

Before starting any Device or Normal Noncachable accesses, the Slots Unit ensures that:

- For reads, the Store Buffer is empty
- No other memory request is currently pending in the Slots Unit.

All Device or Normal Noncachable memory accesses use the imprecise abort mechanism.

This means that Device transfers are considered as bufferable, non-merging for the Store Buffer, and that the Slots Unit does not allow Device requests reordering.

In case of a locked transfer (SWAP instruction) or an exclusive write (STREX variants), the Slots Unit ensures that:

- The Store Buffer is empty.
- The Line Fill Buffers and Eviction Buffer are empty
- No other memory request is currently pending in the Slots Unit.

Noncachable locked and exclusive write memory transfers use the precise abort mechanism.

In all cachable-coherent cases (hit or miss) the cache line ends up in the cache in Modified MESI state.

7.2.3 External Aborts handling

The L1 data cache handles two types of external abort depending on the memory region of the access:

- All Strongly Ordered accesses use the precise abort mechanism.

- All noncachable locked accesses and exclusive stores use the precise abort mechanism.

- All cachable, Device and normal noncachable memory requests use the imprecise abort mechanism. For example, an abort returned on a read miss (issuing a linefill) is flagged as imprecise.

7.2.4 Micro TLB

The micro TLB is designed so that physical addresses can be accessed in time to be able to perform the requested cache access.

Using the address value generated by the core, this produces the physical address and page attributes in the same clock cycle. The page attribute information is used to prevent any cache lookup for noncachable accesses.

This block is organized as an 8-entry Virtual Address based Micro-TLB which does alignment checks and Access Permission checks. It is also connected to the Main TLB block which handles Micro-TLB misses.

7.2.5 Cache arbiter

The Cache Arbiter controls accesses to the Data RAMs, Dirty RAMs, and Tag RAMs. It receives requests from the following sources:

- CCB controller requests
- Eviction Buffer filling requests
- read requests
- allocation requests
- Store Buffer requests
- CP15 requests.

Tag RAM

Tag RAMs consist of four ways of up to 512 lines of 22-bit wide RAM organized as follows:

- 20 bits of tag information
- One MESI exclusive bit
- One valid bit.

Dirty RAM

The dirty RAM stores the following information:

- line dirty attribute, one bit
- line inner attributes for evictions, one bit

- line outer attributes for evictions, two bits
- line shared bit attribute, one bit
- locally modified attribute, one bit. This bit indicates if the cache line has been modified by the current processor. This is different from the dirty attribute as a line can be allocated in the cache while already being in dirty state (migratory line).

The dirty RAM array consists of one bank of up to 512 24-bit lines. One line of dirty RAM contains all information for the four cache ways at a given index.

Data RAM

The data RAM array consists of eight banks of up to 2048 lines of 32-bit wide RAM. It contains cache line data that can be read by the MP11 core.

7.2.6 Store buffer

The features of the store buffer are:

- provides four entries with 64-bits of data and 32 bits of physical address for each slot
- merging capability, that is, writes can merge in the store buffer
- read requests can hit in the store buffer. In that case, data is directly sent back to the core
- write allocation support, that is, write misses in write-back write-allocate regions produce cache linefills.

The store buffer redirects write requests to the following blocks:

- Data cache Arbiter for cachable write hits
- Bus Interface Unit (BIU) for noncachable writes.
- Line Fill Buffer for write misses.

Note

If you perform a write that is NCB followed by a write that is CB while the NCB is still in the store buffer and the CB is in the same cache line, the system issues a deadlock event. This sequence of writes is a programming error and the result is Unpredictable. See *c15, Performance Monitor Control Register (PMNC)* on page 3-67 for details of the deadlock event.

7.2.7 Linefill buffers

There are two linefill buffers, 256 bits wide, that is, one for each cache line with 32 byte enables that allow merging with data from the store buffer.

One bit stores any abort encountered during the linefill that prevents the line from being allocated in the cache.

- If the linefill is the result of a write miss (write to write-back write-allocate region), the write is lost and an imprecise abort is reported to the core.
- If a write access to a write-back region (read or write allocate) hits in the aborted linefill buffer, the write is lost and an imprecise abort is reported to the core.
- Sometimes the start of a linefill is suspended to take advantage of a potential full line merging out of the store buffer into the linefill buffer. This prevents unnecessary external reads in cases of write misses.

The data received by a linefill buffer is streamed to the core so that it does not have to wait for linefill completion before getting its data back. In addition, read requests can hit in a linefill buffer if it contains requested data that has not drained yet.

A drain of a linefill buffer to the cache appears to occur in only one clock cycle. The whole line is forwarded to the cache and written in the RAMs at the same time.

7.2.8 DDI buffer

The DDI buffer is only used when the CPU receives a command from the SCU. It receives a whole line from the data cache and then sends it to the SCU through the bus interface. It also signals to the SCU if the line is dirty or not, and forwards the locally modified flag

7.2.9 Eviction buffer

This buffer receives the data from the cache from lines that have been evicted (by cache clean operation or natural eviction). The data cache only handles one dirty bit per line, so no half-line evictions are possible.

It then forwards the 256-bit line and its physical address to the bus interface unit.

7.3 About the Level 1 Instruction Side memory system

The level 1 instruction side memory system is responsible for providing instruction stream to the integer core of MP11. To increase overall performance and to reduce power consumption, it contains the following functionality:

- Branch prediction
- Branch folding
- Instruction caching.

Branch prediction and branch folding take place before instruction cache lookup. Only predicted fetches are performed in the instruction cache.Figure 7-2 shows the dynamic branch prediction and instruction cache lookup blocks.

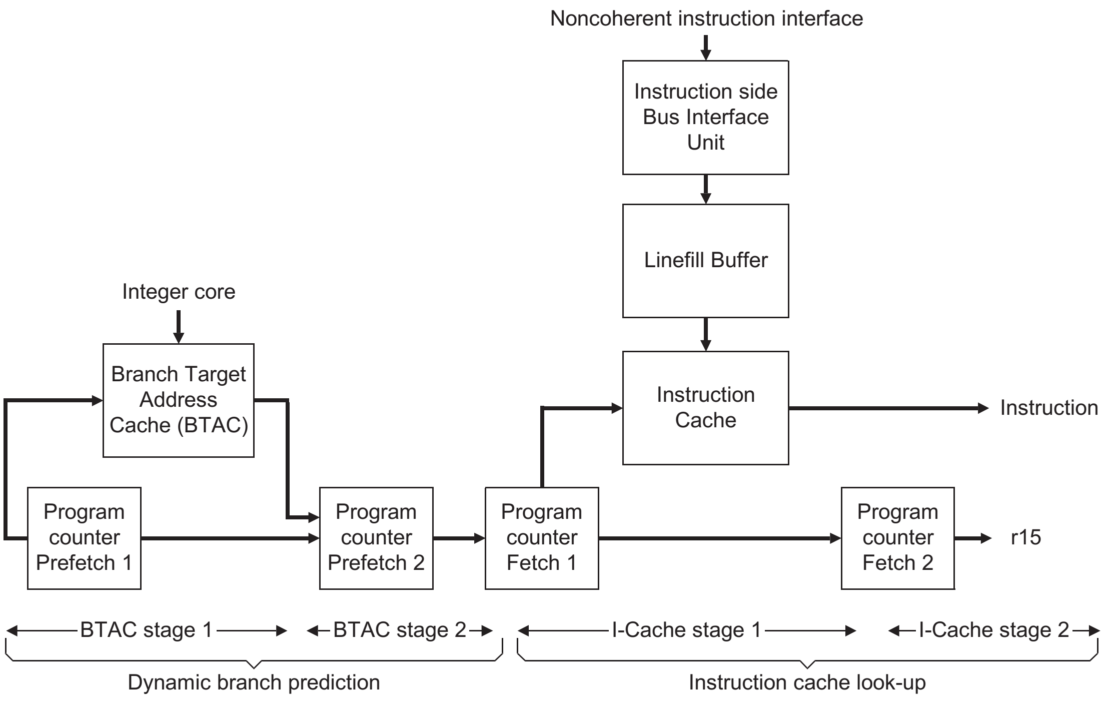

Noncoherent instruction interface
Instruction side Bus Interface Unit
Linefill Buffer
Integer core
Branch Target Address Cache (BTAC)
Instruction Cache
Instruction
Program counter Prefetch 1
Program counter Prefetch 2
Program counter Fetch 1
Program counter Fetch 2
r15
BTAC stage 1
BTAC stage 2
I-Cache stage 1
I-Cache stage 2
Dynamic branch prediction
Instruction cache look-up

Figure 7-2 Level one instruction side level one memory system

7.4 TLB Organization

TLB organization is described in the following sections:

- *Micro TLB*
- *Main TLB*.

7.4.1 Micro TLB

The first level of caching for the page table information is a small MicroTLB of eight entries that is implemented on each of the instruction and data sides. These blocks are implemented in logic, providing a fully associative lookup of the virtual addresses in a cycle for the instruction side. (The Instruction cache is virtually indexed and physically tagged), and in the current cycle for the data side (The Data cache is physically indexed and physically tagged).

The Micro TLB returns the physical address to the cache for the address comparison, and also checks the protection attributes in sufficient time to signal a Data Abort. An additional set of attributes, to be used by the cache line miss handler, is provided by the Micro TLB.

All main TLB related operations affect both the instruction and data Micro TLBs, causing them to be flushed. In the same way, any change of the Context ID Register or of the FCSE PID Register causes the Micro TLBs to be flushed. That is necessary because page table attributes ASID are not stored in Micro TLBs, and because the Micro TLBs work on virtual addresses (VA) but not on modified virtual addresses (MVA).

7.4.2 Main TLB

The main TLB is the second layer in the TLB structure that catches the cache misses from the Micro TLBs. It provides a centralized source for lockable translation entries.

Misses from the instruction and data Micro TLBs are handled by a unified main TLB, that is accessed only on Micro TLB misses. Accesses to the main TLB take a variable number of cycles, according to competing requests between each of the Micro TLBs and other implementation-dependent factors. Entries in the lockable region of the main TLB are lockable at the granularity of a single entry. As long as the lockable region does not contain any locked entries, it can be allocated with non-locked entries to increase overall main TLB storage size.

The main TLB is implemented as a combination of two elements:

- a fully-associative array of eight elements, which is lockable.

- a low-associative (2- way) Tag RAM and Data RAM structure similar to that used in the cache.

Memory access sequence

When an MP11 CPU generates a memory access, the MMU:

1. Performs a lookup for a mapping for the requested virtual address in the relevant instruction or data Micro TLB.
2. If step 1 misses then a lookup for a mapping for the requested modified virtual address (virtual address remapped with FCSE if set) and current ASID in the main TLB is performed.

If no global mapping, or mapping for the currently selected ASID, for the modified virtual address can be found in the TLB then a translation table walk is automatically performed by hardware.

If a matching TLB entry is found then the information it contains is used as follows:

1. The access permission bits and the domain are used to determine if the access is allowed. If the access is not allowed, the MMU signals a memory abort, otherwise the access is enabled to proceed.
2. The memory region attributes are used to control the cache and write buffer, and to determine if the access is cached, non-cached, or device, and if it is shared.
3. The physical address is used for any access to external memory to perform Tag matching for cache entries.

TLB match process

Each TLB entry contains a modified virtual address, a page size, a physical address, and a set of memory properties. Each is marked as being associated with a particular application space, or as global for all application spaces. Register c13 in PC15 determines the currently selected application space. A TLB entry matches if bits [31:N] of the modified virtual address match, where N is log2 of the page size for the TLB entry. It is either marked as global, or the Application Space Identifier (ASID) matched the current ASID. The operating system must ensure that, at most, one TLB entry matches at any time. A TLB can store entries based on the following four block sizes:

Supersections	Consist of 16MB blocks of memory.
Sections	Consist of 1MB blocks of memory.
Large pages	Consist of 64KB blocks of memory.
Small pages	Consist of 4KB blocks of memory.

Supersections, sections and large pages are supported to permit mapping of a large region of memory while using only a single entry in a TLB. If no mapping for an address is found within the TLB, then the translation table is automatically read by hardware and a mapping is placed in the TLB.

Chapter 8
Level 2 Memory System

This chapter describes the Level 2 Memory System. It contains the following sections:

8.1 MPCore Level 2 interface

This section describes the MPCore Level 2 interface in:

- *MPCore Level 2 interface overview*
- *Supported AXI transfers* on page 8-3
- *AXI transaction IDs* on page 8-3
- *Using the STRT instruction* on page 8-4.

8.1.1 MPCore Level 2 interface overview

The ARM11 MPCore processor Level 2 interface consists, by default, of two 64-bit wide AXI bus masters. If the two AXI bus masters have been implemented at implementation time, the ARM11 MPCore processor can still operate as a single bus device by disabling AXI bus master 1 by tying **MASTER1EN** LOW.

Table 8-1 shows the AXI master interface attributes.

Table 8-1 AXI master interface attributes

Attribute	Format
Write Issuing Capability	42, including • eight noncachable writes per MP11 CPU • two evictions per MP11 CPU • 2 DDI evictions.
Read Issuing Capability	16, including • one noncachable read per MP11 CPU • one instruction read per MP11 CPU • two linefill reads per MP11 CPU.
Combined Issuing Capability	58
Write ID Capability	16
Write Interleave Capability	1
Write ID Width	4
Read ID Capability	16
Read ID Width	4

The AXI protocol and meaning of each AXI signal are not described in this document. For a detailed description see *AMBA AXI Protocol v1.0 Specification*.

Supported AXI transfers

ARM11 MPCore master ports generate only a subset of all possible AXI transactions.

For coherent and noncoherent write-back write-allocate transfers the supported transfers are:

- WRAP4 64-bit for read transfers (Linefills)
- INCR4 64-bit for write transfers (Evictions)

For noncoherent noncachable transfers:

- INCRN (N:1-16) 32-bit read transfers
- INCRN (N:1-8) 64-bit read transfers
- INCR1 8-bit, 16-bit, 32-bit and 64-bit read transfers
- INCRN (N:1-16) 32-bit write transfers
- INCRN (N:1-8) 64-bit write transfers
- INCR1 8-bit, 16-bit, 32-bit and 64-bit write transfers
- INCR1 8-bit, 16-bit, 32-bit, 64-bit exclusive read transfers
- INCR1 8-bit, 16-bit, 32-bit, 64-bit exclusive write transfers
- INCR1 32-bit read/write (locked) for swap.

The following points apply to AXI transfers:

- Wrap burst are only read transfers, 64-bit, 4 beats
- Incr single can be any size for read or write
- Incr burst (more than one beat) are only 32-bit or 64-bit
- No transfer is marked as static
- Write transfers with all byte strobes low can occur.

8.1.2 AXI transaction IDs

The AXI ID signal is encoded as follows:

- Bits 0 and 1 define the index of the requesting CPU:
  - 2'b00 for CPU0
  - 2'b01 for CPU1
  - 2'b10 for CPU2
  - 2'b11 for CPU3.
- Bit 2 and 3 for a read transaction define the transfer type:
  - 2'b00 for noncachable data read
  - 2'b01 for instruction fetch
  - 2'b10 for data linefills

    — 2'b11 for data linefills.

- Bit 2 and 3 for a write transaction define the transfer type:
    - 2'b00 for noncachable data write
    - 2'b01 for DDI eviction
    - 2'b10 for data evictions
    - 2'b11 for data evictions.

The arbitration for transaction ordering on AXI masters is round robin among the requesting CPUs. For each CPU the data side has priority over the instruction side.

8.1.3 Using the STRT instruction

Table 8-2 shows core modes and corresponding **APROT** values.

Table 8-2 Core mode and APROT values

User or Privileged core mode	Type of access	Value of APROT
User	Cachable read access	User
Privileged		Privileged
User	Noncachable read access	User
Privileged		Privileged
-	Cachable write access	Always marked as Privileged
User	Noncachable write access	User
Privileged	Noncachable write access	Privileged, except when using `STRT`

Take particular care with Noncachable write accesses when using the `STRT` instruction. To put the correct information on the external bus ensure one of the following:

- the access is to Strongly-ordered memory.

  This ensures that the STRT instruction does not merge in the store buffer.

- the access is to Device memory.

  This ensures that the STRT instruction does not merge in the store buffer.

- a drain write buffer command is issued before the STRT and after the STRT.

  This prevents an STRT from merging into an existing slot at the same 64-bit address, or being merged with another write at the same 64-bit address.

8.2 L2 exclusive mode

You can put each MP11 CPU into L2 exclusive mode by writing to CP15 register (see *c1, Auxiliary Control Register* on page 3-33).

When in L2 exclusive mode, the MPCore signals evictions to the L2 with **AWUSER0** and **AWUSER1**. Table 8-3 shows the meanings of these pins.

Table 8-3 AWUSER pins and meanings

Bits	Meaning
AWUSERx[6]	The current eviction transfer is a clean transfer
AWUSERx[5]	The current transfer is an eviction
AWUSERx[4:1]	The attributes of the written address in the L1 memory system 0000 Strongly ordered 0001 Device 0011 Noncacheable 0110 Write through 0111 Write back 1111 Write back, Write allocate These are also called inner attributes.
AWUSERx[0]	Shared attribute

8.3 Synchronization operations

Synchronization primitives are an extremely important part of an SMP system, and must be treated with great care. The primitives are swap instructions, exclusive load instructions, and exclusive store instructions.

These synchronization primitives can be used in coherent or noncachable regions. Coherent regions are defined as Cachable, Write-Back Write-Allocate shared memory regions, when the SMP bit is set (see *c1, Auxiliary Control Register* on page 3-33).

Coherent synchronization primitives

These operations are cached in the level 1 memory system, and the SCU maintains coherency inside the caches and resolves hazards.

Exclusive synchronization primitives use the internal monitor block which is available in all data caches of the ARM11 MPCore processor. An external exclusive monitor is not needed for coherent exclusive primitives to work.

Noncachable synchronization primitives

These are treated as in a nonmultiprocessing system. For swap instructions, the SCU ensures that locked transfers are atomic.

Exclusive transfers are treated as normal transfers, and an exclusive monitor must be externally implemented to monitor exclusive transactions. Such transfers are not detailed in this manual.

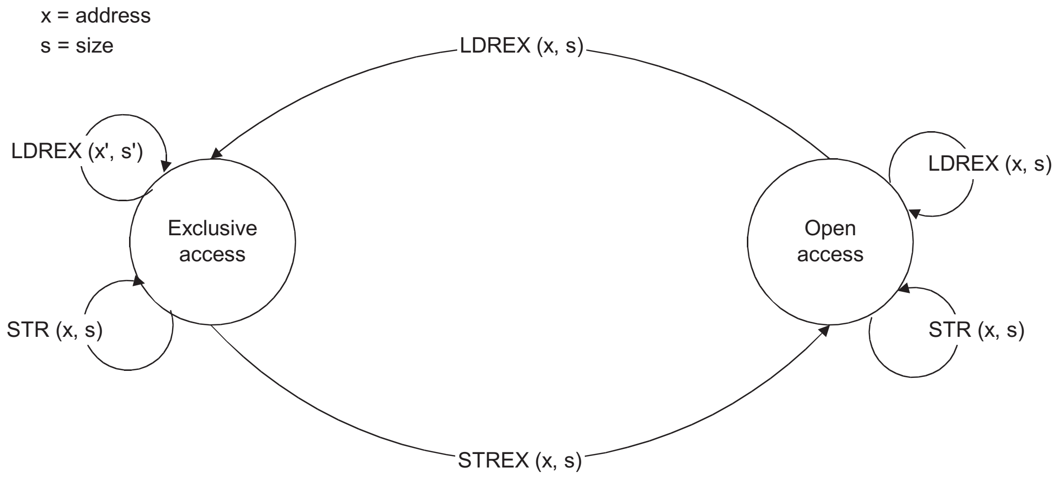
At the SCU level all noncoherent swaps and exclusive transfers are sent to master port 0.

8.3.1 Exclusive loads and stores

Exclusive loads and stores are a way to execute semaphores where the load and store operations are nonatomic. This enables operations to be performed on the loaded data before storing it.

The system guarantees that if the data that has been previously loaded has been modified by another CPU, the store fails and the load-store sequence must be retried.

This behavior is checked using a monitor. This monitor is internal to the CPU. Figure 8-1 on page 8-7 shows the behavior of the exclusive monitor state machine.



STREX value	Result
STREX (Tagged_address, Tagged_size)	OKAY
STREX (Overlap)	XFAIL
STREX (!Overlap)	XFAIL
Cache Line = (Tagged_address)	The cache line is cleaned and invalidated

Figure 8-1 Exclusive monitor state machine

Chapter 9
MPCore Private Memory Region

This chapter describes the remappable memory region used to internally access the private MPCore peripherals, the Interrupt Distributor, the MP11 CPU Interfaces, the Timers and Watchdog, and the *Snoop Control Unit* (SCU). It contains the following section:

- *About the MPCore private memory region* on page 9-2.

9.1 About the MPCore private memory region

Most of the MP11 CPU control operations are through CP15 instructions. These operations are not applicable to ARM11 MPCore processor global control. ARM11 MPCore processor global control and peripherals must be accessed through memory-mapped transfers. To reduce the complexity of the system and hide these transactions from the L2 memory system, all registers accessible by all MP11 CPUs within MPCore are grouped into two contiguous 4KB pages accessed through a dedicated internal bus. These pages are relocatable because the base address is defined using pins **PERIPHBASE[18:0]**.

Any transaction to addresses of the form `{PERIPHBASE[18:0], 13'address_low_bits}` is redirected to the MPCore private memory region.

Table 9-1 shows register addresses for the ARM11 MPCore processor relative to this base address.

Table 9-1 MPCore private memory region

Offset	Peripheral	Description
`0x0000 - 0x00FF`	SCU registers	*SCU-specific registers* on page 9-3
`0x0100 - 0x01FF`	CPU interrupt interfaces (identified by CPU transaction ID)	See *CPU Interfaces* on page 10-8.
`0x0200 - 0x2FF`	CPU 0 interrupt interface (aliased for debug purposes)	
`0x0300 - 0x03FF`	CPU 1 interrupt interface (aliased for debug purpose)	
`0x0400 - 0x04FF`	CPU 2 interrupt interface (aliased for debug purposes)	
`0x0500 - 0x05FF`	CPU 3 interrupt interface (aliased for debug purposes)	

Table 9-1 MPCore private memory region (continued)

Offset	Peripheral	Description
`0x0600 - 0x06FF`	CPU timer and watchdog (identified by CPU transaction ID)	See *Timer and Watchdog blocks* on page 10-23.
`0x0700 - 0x07FF`	CPU0 timer and watchdog	
`0x0800 - 0x08FF`	CPU1 timer and watchdog	
`0x0900 - 0x09FF`	CPU2 timer and watchdog	
`0x0A00 - 0x0AFF`	CPU3 timer and watchdog	
`0x0b00 - 0x0FFF`	Reserved	Any access to this region causes a DECERR abort exception.
`0x1000 - 0x1FFF`	Global Interrupt distributor	See *Interrupt Distributor* on page 10-4.

9.1.1 SCU-specific registers

Table 9-2 shows the SCU-specific registers. Addresses are given relative to the base address of the region for the SCU in the private memory region memory map. All SCU registers are byte accessible.

Table 9-2 SCU register definition

Offset	Name	Reset value	Type	Description
`0x00`	Control Register	`0x00000000`	R/W	See *SCU Control Register* on page 9-4.
`0x04`	Configuration Register	Implementation Defined	RO	See *SCU Configuration Register* on page 9-5
`0x08`	SCU CPU Status	-	R/W	See *SCU CPU Status Register* on page 9-6.
`0x0C`	Invalidate all	-	WO	See *SCU Invalidate All Register* on page 9-8.
`0x10`	Performance Monitor Control Register	`0x00000000`	R/W	See *Performance Monitor Control Register* on page 9-8.
`0x14`	Monitor Counter Events 0	`0x00000000`	R/W	See *Performance monitor event registers* on page 9-11
`0x18`	Monitor Counter Events 1	`0x00000000`	R/W	

Table 9-2 SCU register definition (continued)

Offset	Name	Reset value	Type	Description
0x1C	Monitor Counter 0	`0x00000000`	R/W	See *Count registers, MN0-MN7* on page 9-13
0x20	Monitor Counter 1	`0x00000000`	R/W	
0x24	Monitor Counter 2	`0x00000000`	R/W	
0x28	Monitor Counter 3	`0x00000000`	R/W	
0x2C	Monitor Counter 4	`0x00000000`	R/W	
0x30	Monitor Counter 5	`0x00000000`	R/W	
0x34	Monitor Counter 6	`0x00000000`	R/W	
0x38	Monitor Counter 7	`0x00000000`	R/W	
`0x3C` - `0xFC`	Reserved	-	-	RAZ

9.1.2 SCU Control Register

The SCU Control Register enables the SCU and controls its behavior. It must be accessed using a read-modify-write sequence. Figure 9-1 shows the format of the SCU Control Register.

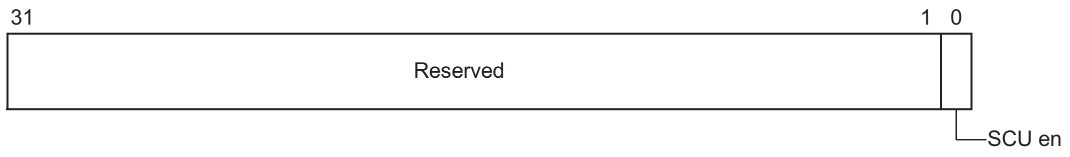


Figure 9-1 SCU Control Register format

Table 9-3 shows the SCU Control Register bit assignments.

Table 9-3 SCU Control Register bit assignments

Bits	Field	Description
[31:1]	Reserved	SBZ
[0]	SCU En	1 SCU is enabled, coherency is maintained between MP11 CPUs Level 1 data side caches. 0 SCU is disabled, coherency is not maintained between MP11 CPUs Level 1 data side caches. In an ARM11 MPCore single CPU configuration, writing to this bit has no effect and it is always read as zero.

9.1.3 SCU Configuration Register

The SCU Configuration Register is read-only. Figure 9-2 shows the format for this register.

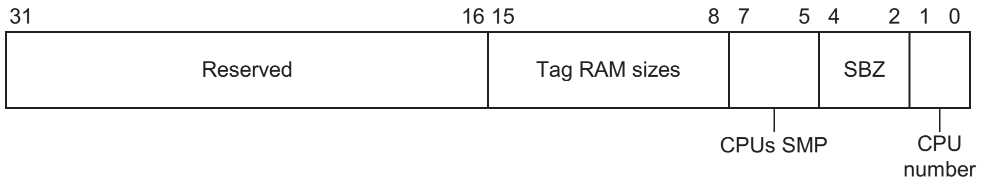


Figure 9-2 SCU Configuration Register format

Table 9-4 shows the SCU Configuration Register bit assignments.

Table 9-4 SCU Configuration Register bit assignments

Bits	Field	Description
[31:15]	Reserved	SBZ.
[15:8]	Tag RAM sizes	Bits [15:14] define MP11 CPU3 tag RAM size if present. Bits [13:12] define MP11 CPU2 tag RAM size if present. Bits [11:10] define MP11 CPU1 tag RAM size if present. Bits [9:8] define MP11 CPU0 tag RAM size. The encoding is as follows 2'b11 reserved 2'b10 64KB cache, 256 indexes per tag RAM 2'b01 32KB cache, 128 indexes per tag RAM 2'b00 16KB cache, 64 indexes per tag RAM.

Table 9-4 SCU Configuration Register bit assignments

Bits	Field	Description
[7:4]	CPUs SMP	Defines which CPUs are in SMP (Symmetric Multi-processing) or AMP (Asymmetric Multi-processing) mode. 1 MP11 CPU is in SMP mode taking part in coherency. 0 MP11 CPU is in AMP mode not taking part in coherency or not present. Bit 7 is for MP11 CPU3 Bit 6 is for MP11 CPU2 Bit 5 is for MP11 CPU1 Bit 4 is for MP11 CPU0.
[3:2]	Reserved	SBZ
[1:0]	CPU Number	Number of CPUs present in the ARM11 MPCore processor 2'b11 4 MP11 CPUs, CPU0-CPU3 2'b10 3 MP11 CPUs, CPU0-CPU2 2'b01 2 MP11 CPUs, CPU0-CPU1 2'b00 1 MP11 CPU, CPU0.

9.1.4 SCU CPU Status Register

This register is a read/write register that specifies the state of the MP11 CPUs. Figure 9-3 on page 9-7 shows the format for the SCU CPU Status Register.

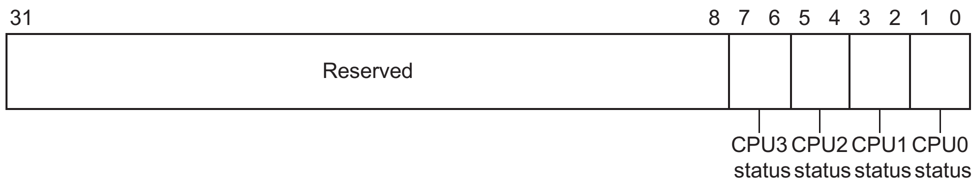


Figure 9-3 SCU MP11 CPU Status Register

Table 9-5 shows the SCU CPU Status Register bit assignments.

Table 9-5 SCU CPU Status Register bit assignments

Bits	Field	Description
[31:8]	Reserved	SBZ
[7:6]	CPU3 status	Status of the MP11 CPU 2'b11: CPU is about to enter (or is in) powered-off mode, or is nonpresent No CCB request is sent to the CPU 2'b10: CPU is about to enter (or is in) dormant mode. No CCB request is sent to the CPU 2'b01: RESERVED 2'b00: Normal mode (Default).
[5:4]	CPU2 status	
[3:2]	CPU1 status	
[1:0]	CPU0 status	

Dormant mode and powered-off mode are controlled by an external power controller. SCU CPU Status Register bits indicate to the external power controller which power domains can be powered down.

Before entering any other power mode than Normal, the MP11 CPU must set its status field to signal to the SCU which mode it is about to enter (so that the SCU can determine if it still can send coherency requests to the CPU). The MP11 CPU then executes a WFI entry instruction. When in WFI state, the PWRCTLOn bus is enabled and signals to the power controller what it should do with power domains.

The SCU CPU Status Register bits are used in conjunction with internal WFI entry signals to generate PWRCTLOn output pins.

The SCU CPU Status Register bits can also be read by a CPU exiting low-power mode to determine its state before executing its reset setup.

MP11 CPUs status fields take PWRCTLIn values at reset, except for nonpresent CPUs. For nonpresent CPUs writing to this field has no effect.

9.1.5 SCU Invalidate All Register

The SCU Invalidate All Register invalidates the tag RAMs on a per CPU and per way basis. This operation is atomic, that is, a write transfer to this address only terminates when all the lines have been invalidated. This register reads as 0.

Figure 9-4 shows the format of this register.

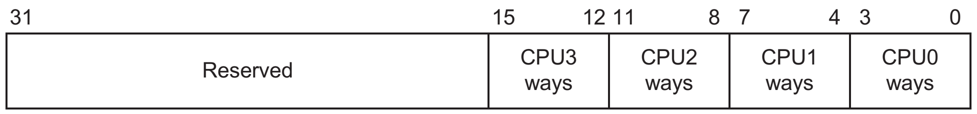


Figure 9-4 SCU Invalidate All Register format

Table 9-6 shows the SCU Invalidate All Register bit assignments.

Table 9-6 SCU Invalidate All Register bit assignment

Bits	Field	Description
[31:16]	Reserved	SBZ
[15:12]	CPU3 ways	Indicates the ways that must be invalidated for MP11 CPU3. Writing to these bits has no effect if the ARM11 MPCore processor has less than four CPUs
[11:8]	CPU2 ways	Indicates the ways that must be invalidated for MP11 CPU2. Writing to these bits has no effect if the ARM11 MPCore processor has less than three CPUs
[7:4]	CPU1 ways	Indicates the ways that must be invalidated for MP11 CPU1. Writing to these bits has no effect if the ARM11 MPCore processor has less than two CPUs
[3:0]	CPU0 ways	Indicates the ways that must be invalidated for MP11 CPU0. Writing to these bits has no effect if the ARM11 MPCore processor has less than two CPUs

9.1.6 Performance Monitor Control Register

The Performance Monitor Control Register controls the operation of the Count Registers. This register:

- Detects which counter overflowed.
- Enables and disables interrupt reporting.
- Resets all counters to zero.
- Enables the entire performance monitoring mechanism.

The number of counters available in the design depends on the number of MP11 CPUs in the ARM11 MPCore processor. Table 9-7 shows the relationship of the number of MP11 CPUs and the number of counters.

Table 9-7 MP11 CPUs and counters

Number of MP11 CPUs	Number of counters available
4	Counters MN0-MN7
3	Counters MN0-MN5
2	Counters MN0-MN3
1	Counters MN0 and MN1

Figure 9-5 shows the format of the SCU Performance Monitor Control Register.

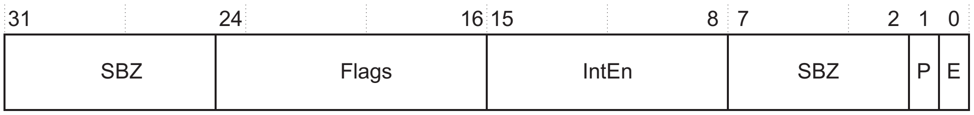


Figure 9-5 SCU Performance Monitor Control Register format

Table 9-8 shows the Performance Monitor Control Register bit assignments.

Table 9-8 Performance Monitor Control Register bit assignments

Bit	Name	Function
[31:24]	-	SBZ/RAZ
[23:16]	Flags	Overflow/Interrupt flags Identifies which counter overflowed. Bit 23 Count register MN7 overflow flag, if available. Bit 22 Count register MN6 overflow flag, if available. Bit 21 Count register MN5 overflow flag, if available. Bit 20 Count register MN4 overflow flag, if available. Bit 19 Count register MN3 overflow flag, if available. Bit 18 Count register MN2 overflow flag, if available. Bit 17 Count register MN1 overflow flag, if available. Bit 16 Count register MN0 overflow flag. Writing 1 to a bit has the effect of clearing it.
[15:8]	IntEn	Interrupt enable Enables and disables interrupt reporting for each counter, when available. Bit 15 enables interruption for counter MN7 when set. Bit 14 enables interruption for counter MN6 when set. Bit 13 enables interruption for counter MN5 when set. Bit 12 enables interruption for counter MN4 when set. Bit 11 enables interruption for counter MN3 when set. Bit 10 enables interruption for counter MN2 when set. Bit 9 enables interruption for counter MN1 when set. Bit 8 enables interruption for counter MN0 when set. IntEn bits for counters that are not available always read as 0.
[7:2]	Reserved	SBZ
[1]	P	Reset all count registers when this bit is written to 1.
[0]	E	Enable bit 1 = all counters enabled 0 = all counters disabled.

If an interrupt is generated by the Performance Monitor Control Register, the relevant bits of **PMUIRQ** are asserted:

- **PMUIRQ[11]** is asserted on overflow from counter MN7.
- **PMUIRQ[10]** is asserted on overflow from counter MN6.

- **PMUIRQ[9]** is asserted on overflow from counter MN5.
- **PMUIRQ[8]** is asserted on overflow from counter MN4.
- **PMUIRQ[7]** is asserted on overflow from counter MN3.
- **PMUIRQ[6]** is asserted on overflow from counter MN2.
- **PMUIRQ[5]** is asserted on overflow from counter MN1.
- **PMUIRQ[4]** is asserted on overflow from counter MN0.

9.1.7 Performance monitor event registers

Event registers PME0 and PME1 identify the source of the event for counters MN0-MN7. Table 9-9 shows the events and their definitions.

Table 9-9 Event definitions

Event number	Event source	Event definition
0	-	Counter disabled
1	CPU0	The corresponding CPU has requested a coherent linefill that misses in all the other CPUs. The request is sent to external memory.
2	CPU1	
3	CPU2	
4	CPU3	
5	CPU0	The corresponding CPU has requested a coherent linefill that hits in another CPU. The linefill is fetched directly from the relevant CPU cache.
6	CPU1	
7	CPU2	
8	CPU3	
9	CPU0	The corresponding CPU was expected to have a coherent line in its cache but answers nonpresent.
10	CPU1	
11	CPU2	
12	CPU3	
13	-	Line migration. A line is directly transferred from one CPU to another on a linefill request instead of switching to SHARED.
14	Master0	The read port of the corresponding master port is busy.
15	Master1	

Table 9-9 Event definitions (continued)

Event number	Event source	Event definition
16	Master0	The write port of the corresponding master port is busy.
17	Master1	
18	-	A read transfer is sent to the external memory.
19	-	A write transfer is sent to the external memory.
20-30	-	-
31	Cycle count	The counter increments on each cycle.

Performance Monitor Event Register 0

Figure 9-6 shows the format of Performance Monitor Event Register 0.

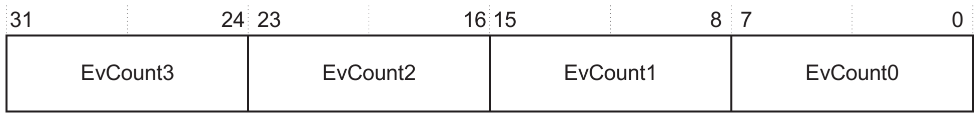


Figure 9-6 Performance Monitor Event Register 0 bit format

Table 9-10 shows the Performance Monitor Event Register 0 bit assignments.

Table 9-10 Performance Monitor Event Register 0 bit assignments

Bit	Name	Function
[31:24]	EvCount3	Identifies the event for counter MN3. In configurations with less than two MP11 CPUs these bits always read as 0.
[23:16]	EvCount2	Identifies the event for counter MN2. In configurations with less than two MP11 CPUs these bits always read as 0.
[15:8]	EvCount1	Identifies the event for counter MN1.
[7:0]	EvCount0	Identifies the event for counter MN0.

Performance Monitor Event Register 1

Figure 9-6 on page 9-12 shows the format of Performance Monitor Event Register 1.

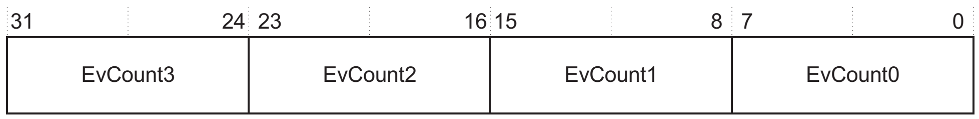


Figure 9-7 Performance Monitor Event Register 1 bit format

Table 9-11 shows the Performance Monitor Event Register 1 bit assignments.

Table 9-11 Performance Monitor Event Register 1 bit assignments

Bit	Name	Function
[31:24]	EvCount7	Identifies the event for counter MN7. In configurations with less than four MP11 CPUs these bits always read as 0.
[23:16]	EvCount6	Identifies the event for counter MN6. In configurations with less than four MP11 CPUs these bits always read as 0.
[15:8]	EvCount5	Identifies the event for counter MN5. In configurations with less than three MP11 CPUs these bits always read as 0.
[7:0]	EvCount4	Identifies the event for counter MN4. In configurations with less than three MP11 CPUs these bits always read as 0

9.1.8 Count registers, MN0-MN7

The value in the count registers is 0 at reset. The count registers are up registers. A count register is incremented by 1 when the corresponding event is detected. Registers MN0-MN7 are read accessible and write accessible. The number of available count registers depends on the number of MP11 CPUs instantiated in the ARM11 MPCore processor as shown in *MP11 CPUs and counters* on page 9-9.

Chapter 10
MPCore Distributed Interrupt Controller

This chapter describes the Distributed Interrupt Controller. The Distributed Interrupt Controller collates interrupts from a number of sources and arbitrates between them. This chapter contains the following sections:

10.1 About the Distributed Interrupt Controller

The Distributed Interrupt Controller collates interrupts from a large number of sources. It provides:

- masking of interrupts
- prioritization of the interrupts
- distribution of the interrupts to the target MP11 CPUs
- tracking the status of interrupts
- generation of interrupts by software.

The Distributed Interrupt Controller is a single functional unit that is placed in the system alongside MP11 CPUs. This enables the number of interrupts supported in the system to be independent of the MP11 CPU design.

The Distributed Interrupt Controller is memory-mapped. It is accessed by the MP11 CPUs using a private interface through the SCU. You can access the Distributed Interrupt Controller using **PERIPHBASE** (see *About the MPCore private memory region* on page 9-2).

10.1.1 Distributed Interrupt Controller Clock frequency

The Distributed Interrupt Controller logic is clocked at half the frequency of the MPCore CPUs because of power and area considerations. Reducing clock speed reduces dynamic power consumption. The lower clock speed requires less pipelining in the design. This means that the overall impact of the reduced clock speed on the Distributed Interrupt Controller is kept to a minimum.

Note

As a consequence, the minimum pulse width of signals driving external interrupt lines is two CPU clock cycles.

10.2 Terminology

For the purposes of this document, interrupts are either asserted or nonasserted. An asserted interrupt is one that is signalling that it is ready to be processed, or is being processed. The transition of an interrupt from the nonasserted state to the asserted state is described as the assertion of the interrupt. The transition of an interrupt from the asserted state to the nonasserted state is described as the clearing of the interrupt.

From the point of view of an MP11 CPU, an interrupt can be:

Inactive An Inactive interrupt is one that is nonasserted, or which in a multi-processing environment has been completely processed by that MP11 CPU but can still be either Pending or Active in some of the MP11 CPUs to which it is targeted, and so might not have been cleared at the interrupt source.

Pending A Pending interrupt is one that has been asserted, and for which processing has not started on that MP11 CPU.

Active An Active interrupt is one that has been started on that MP11 CPU, but processing is not complete.

An interrupt can be Pending and Active at the same time. This can happen in the case of edge triggered interrupts, when the interrupt is asserted while the MP11 CPU has not finished handling the first occurrence. For level-sensitive interrupts it can only happen if software triggers it. See *Interrupt configuration registers, 0xC00-0xC3C* on page 10-16.

Pre-emption An Active interrupt can be pre-empted when a new interrupt of higher priority interrupts MP11 CPU interrupt processing. For the purpose of this document, an Active interrupt may be either running if it is actually being processed, or pre-empted.

The Distributed Interrupt Controller consists of:

Interrupt Distributor

The Interrupt Distributor handles interrupt detection and interrupt prioritization.

CPU interfaces

There is one CPU interface per MP11 CPU. The MP11 CPU interfaces handle interrupt acknowledgement, interrupt masking, and interrupt completion acknowledgement.

10.3 Interrupt Distributor

The Interrupt Distributor centralizes all interrupt sources for the ARM11 MPCore processor before dispatching the highest priority ones to each individual MP11 CPU.

All interrupt sources are identified by a unique ID. All interrupt sources have their own configurable priority and list of targeted CPUs, that is, a list of CPUs to which the interrupt is sent when triggered by the Interrupt Distributor.

Interrupt sources are of the following types:

Interprocessor interrupts (IPI)

Each MP11 CPU has private interrupts, ID0-ID15, that can only be triggered by software. These interrupts are aliased so that there is no requirement for a requesting MP11 CPU to determine its own ID when it deals with IPIs. The priority of an IPI depends on the receiving CPU, not the sending CPU.

Private timer and/or watchdog interrupts.

Each MP11 CPU has its own private timer that can generate interrupts, using ID29 and ID30.

A legacy nIRQ pin

In legacy IRQ mode the legacy nIRQ pin, on a per CPU basis, bypasses the Interrupt Distributor logic and directly drives interrupt requests into the MP11 CPU.

When an MP11 CPU uses the Distributed Interrupt Controller (rather than the legacy pin in the legacy mode) by enabling its own CPU interface, the legacy nIRQ pin is treated like other interrupt lines and uses ID31.

Hardware interrupts

Hardware interrupts are triggered by programmable events on associated interrupt input lines. MP11 CPUs can support up to 224 interrupt input lines. The interrupt input lines can be configured to be edge sensitive (posedge) or level sensitive (high level). Hardware interrupts start at ID32.

10.3.1 Interrupt Distributor overview

The Interrupt Distributor holds the list of Pending interrupts for each CPU, and then selects the highest priority interrupt before issuing it to the CPU interface. Interrupts of equal priority are resolved by selecting the lowest ID.

The Interrupt Distributor consists of a register-based list of interrupts, their priorities and activation requirements (CPU targets). In addition the state of each interrupt on each CPU is held in the associated state storage.

The prioritization logic is physically duplicated for each CPU to enable the selection of the highest priority for each CPU.

The Interrupt Distributor holds the central list of interrupts, processors and activation information, and is responsible for triggering software interrupts to processors.

The CPU Interface acknowledges interrupts and changes interrupt priority masks.

The Interrupt Distributor also contains for each interrupt the software model (1-N or N-N) which that interrupt uses:

- With the 1-N model, an interrupt that is taken on any CPU clears the Pending status on all CPUs.
- With the N-N model, all CPUs receive the interrupt independently. The Pending status is cleared only for the CPU that takes it, not for the other CPUs.

Note

Even with the 1-N software model, two CPUs might still take the same interrupt if the CPU acknowledgements are within a few cycles of each other.

The Interrupt Distributor transmits to the CPU Interfaces their highest Pending interrupt. It receives back the information that the interrupt has been acknowledged, and can then change the status of the corresponding interrupt. The CPU Interface also transmits *End of Interrupt Information* (EOI), which enables the Interrupt Distributor to update the status of this interrupt from Active to Inactive.

Figure 10-1 on page 10-6 shows the main blocks of the Interrupt Distributor.

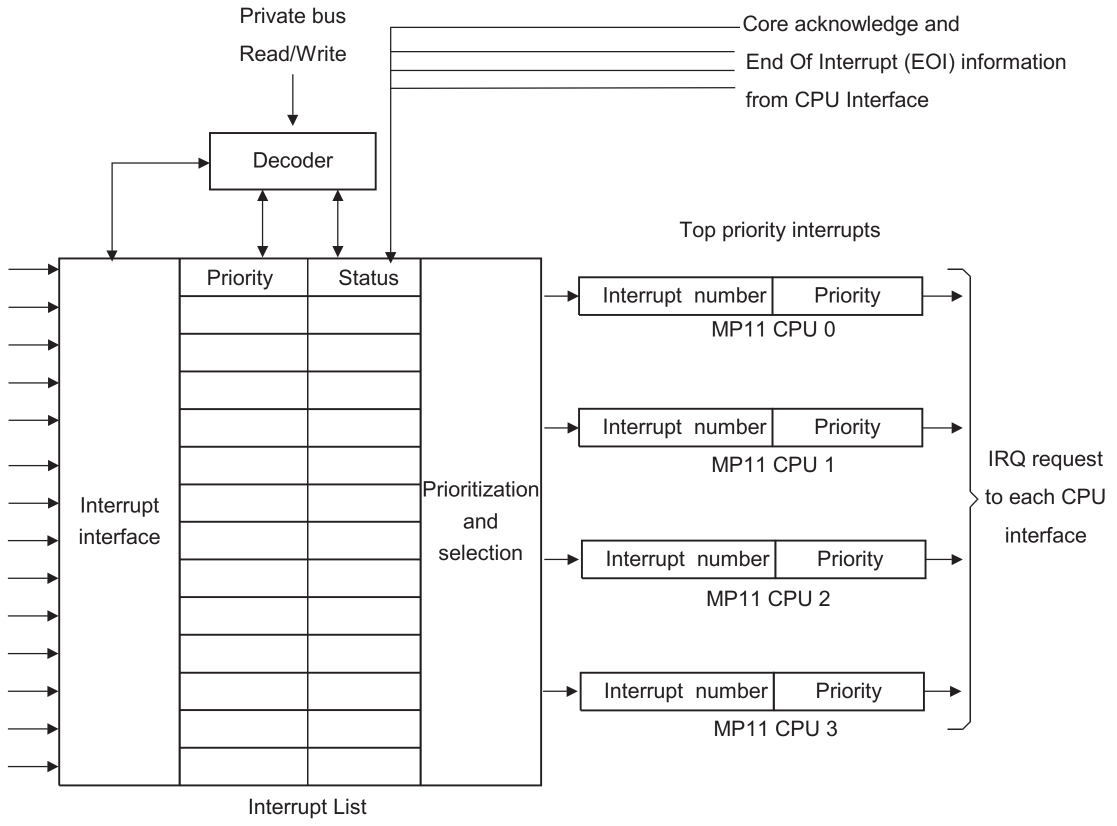


Figure 10-1 Interrupt Distributor block diagram

10.3.2 Behavior of the Interrupt Distributor

When the Interrupt Distributor detects an interrupt assertion, it sets the status of the interrupt for the targeted MP11 CPUs to Pending. Level-triggered interrupts cannot be marked as Pending if they are Active for at least one MP11 CPU.

For each MP11 CPU the prioritization and selection block searches for the Pending interrupt with the highest priority. This interrupt is then sent with its priority to the CPU Interface.

Note

Priority zero is the highest priority. The lowest priority is priority `0xF`.

When multiple Pending interrupts have the same priority, the selected interrupt is the one with lowest ID. If there are multiple Pending software interrupts with the same ID, the lowest MP11 CPU source is selected.

The CPU Interface returns information to the Distributor when the CPU acknowledges (Pending to Active transition) or clears an interrupt (Active to Inactive transition). With the given interrupt ID, the Interrupt Distributor updates the status of this interrupt according to the information sent by the CPU Interface.

When an interrupt is triggered by the Software Interrupt Register or the Set-pending Register, the status of that interrupt for the targeted CPU or CPUs is set to Pending. This interrupt then has the same behavior as a hardware interrupt. The distributor does not differentiate between software and hardware triggered interrupts.

10.4 CPU Interfaces

The CPU interfaces are slaves to the MP11 CPUs. The MP11 CPU interfaces handle interrupt priority masking and pre-empted interrupts.

A Pending interrupt is only accepted if its priority is higher than the priority mask and is also higher than the priority of the highest priority active interrupt Active on that MP11 CPU. If a Pending interrupt is accepted, the effect is that an interrupt request is made to the MP11 CPU for interrupt exception entry.

If the MP11 CPU then reads its Interrupt Acknowledge Register, the CPU interface records the priority of this interrupt and marks it as Active in the Interrupt Distributor for that MP11 CPU.

If the interrupt is cleared before the MP11 CPU reads its Interrupt Acknowledge Register, for example because of a priority mask change or a write to the Interrupt Pending Clear Register, the MP11 CPU gets the interrupt ID value 1023, indicating a spurious interrupt.

The interrupt Active to Inactive transition is triggered by an MP11 CPU writing the completed Interrupt ID in its End of Interrupt Register.

10.5 Interrupt Distributor registers

Table 10-1 shows the addresses of the registers in the Interrupt Distributor. Addresses are given relative to the base address of the region for the Interrupt Distributor defined by the private memory region memory map. See Table 9-1 on page 9-2. All Interrupt Distributor registers are byte accessible.

Table 10-1 Distributed Interrupt Controller programmer's model

Register address	Type	Reset value	Function	Description
`0x000`	R/W	`0x00000000`	Interrupt Distributor Control Register	See *Interrupt Distributor Control Register, 0x000* on page 10-11.
`0x004`	RO	-	Interrupt Controller Type Register	See *Interrupt Controller Type Register, 0x004* on page 10-11.
`0x008-0x0FC`	....	....	....	Reserved
`0x100`	R/W	`0x0000FFFF`	Interrupt Set-enable Registers ID0-ID31	See *Interrupt Clear-enable and Set-enable registers, 0x100-0x11C and 0x180-0x19C* on page 10-12.
`0x104-0x11C`		`0x00000000`	Interrupt Set-enable Registers ID32 and upwards	
`0x120-0x17C`	....	....	....	Reserved
`0x180`	R/W	`0x0000FFFF`	Interrupt Clear-enable Registers ID0-ID31	See *Interrupt Clear-enable and Set-enable registers, 0x100-0x11C and 0x180-0x19C* on page 10-12.
`0x184-0x19C`		`0x00000000`	Interrupt Clear-enable Registers ID32 and upwards	
`0x1A0-`0x1F C	....	....	....	Reserved
`0x200-0x21C`	R/W	`0x00000000`	Interrupt Set-pending Registers	See *Interrupt Clear-pending and Set-pending registers, 0x200-0x21C and 0x280-0x29C* on page 10-13.
`0x220-0x27C`	....	....	....	Reserved.
`0x280-0x29C`	R/W	`0x00000000`	Interrupt Clear-pending Registers	See *Interrupt Clear-pending and Set-pending registers, 0x200-0x21C and 0x280-0x29C* on page 10-13.
`0x2A0-`0x2F C	....	....	....	Reserved

Table 10-1 Distributed Interrupt Controller programmer's model (continued)

Register address	Type	Reset value	Function	Description
`0x300-0x31C`	RO	`0x00000000`	Interrupt Active Bit Registers	See *Active Bit Registers, 0x300-0x31C* on page 10-14
`0x320-0x3FC`	....	....	....	Reserved
`0x400-0x4FC`	R/W	`0x00000000`	Interrupt Priority Registers	See *Interrupt priority registers, 0x400-0x4FC* on page 10-14
`0x500-0x7FC`	....	....	....	Reserved
`0x800-0x8FC`	R/W	`0x00000000`[a]	Interrupt CPU targets Registers	See *Interrupt CPU targets registers, 0x800-0x8FC* on page 10-15
`0x900-0xBFC`	....	....	....	Reserved
`0xC00`	R/W	`0xAAAAAAAA`	Interrupt Configuration Registers, ID0-ID15	See *Interrupt configuration registers, 0xC00-0xC3C* on page 10-16
`0xC04`		`0x28000000`	Interrupt Configuration Registers, ID29-ID31	
`0xC08-0xC3C`		`0x00000000`	Interrupt Configuration Registers, ID32 and upwards	
`0xC40-0xEFC`	....	....	....	Reserved
`0xF00`	WO	-	Software Interrupt Register	See *Software Interrupt Register, 0xF00* on page 10-17
`0xF00-0xFDC`	....	....	....	Reserved
`0xFE0`	RO	`0x90`	Peripheral Identification Register 0	Reserved for future compatibility
`0xFE4`	RO	`0x13`	Peripheral Identification Register 1	
`0xFE8`	RO	`0x04`	Peripheral Identification Register 2	
`0xFEC`	RO	`0x00`	Peripheral Identification Register 3	
`0xFF0`	RO	`0x0D`	PrimeCell Identification Register 0	
`0xFF4`	RO	`0xF0`	PrimeCell Identification Register 1	
`0xFF8`	RO	`0x05`	PrimeCell Identification Register 2	
`0xFFC`	RO	`0xB1`	PrimeCell Identification Register 3	

a. Except for address `0x81C` . See *Interrupt CPU targets registers, 0x800-0x8FC* on page 10-15.

All registers not described in Table 10-1 on page 10-9 are Reserved and read as zero. Writing to these registers has no effect.

10.5.1 Interrupt Distributor Control Register, 0x000

Figure 10-2 shows the format of Interrupt Distributor Control Register.

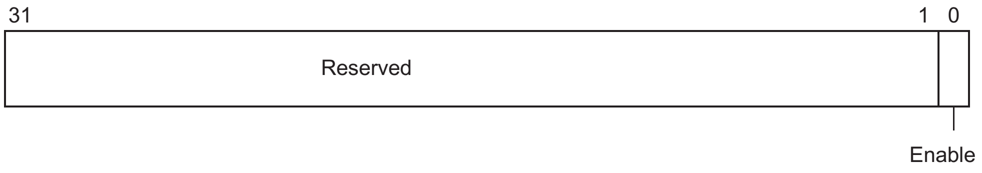


Figure 10-2 Interrupt Distributor Control Register format

Bit 0 of the control register is a global interrupt controller enable. If the value of Bit 0 is 0, no interrupts at all are sent to the CPU Interfaces.

10.5.2 Interrupt Controller Type Register, 0x004

Figure 10-3 shows the format of Interrupt Controller Type Register.

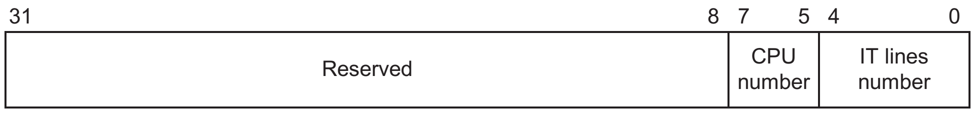


Figure 10-3 Interrupt Controller Type Register format

Table 10-2 shows the Interrupt Distributor Controller Type Register bit assignments.

Table 10-2 Interrupt Controller Type Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved
[7:5]	CPU number	Encoding is: b000 MPCore contains 1 MP11 CPU. b001 MPCore contains 2 MP11 CPUs. b010 MPCore contains 3 MP11 CPUs. b011 MPCore contains 4 MP11 CPUs. b1xx: Reserved for future extensions.
[4:0]	IT lines number	Encoding is: b00000 No external interrupt input lines. (32 interrupt ID support)[a] b00001: 32 external interrupt input lines (64 interrupt ID support). … b00111 224 external interrupt input lines (256 interrupt ID support) All others are Reserved for future extension.

a. Interrupt IDs 0-31 are reserved for software and MP11 CPU private interrupts.

10.5.3 Interrupt Clear-enable and Set-enable registers, 0x100-0x11C and 0x180-0x19C

These registers control an enable bit for each interrupt in the Interrupt Distributor. There can be up to eight Clear-enable and eight Set-enable registers.

For a dedicated interrupt, if its enable bit is set to 1, the interrupt is transmitted to the targeted CPUs if it is Pending. If the bit is set to 0, that interrupt is never sent to any CPU. The enable bit, when set to 0, does not prevent an edge-triggered interrupt from becoming Pending. The enable bit, when set to 0, does prevent a level sensitive interrupt from becoming Pending only if asserted by the hardware pin, **INT**.

Writing a 1 to a bit in the Clear-enable register means that the corresponding interrupt enable bit is set to 0.

Writing a 1 to a bit in the Set-enable register means that the corresponding interrupt enable bit is set to 1.

The values read in Clear-enable and Set-enable registers for the same range of interrupts are the same and represent current enabled interrupts.

For example, if you want to set the enable bit for interrupt ID 119, assuming that the Interrupt Controller Configuration supports at least 128 IDs, you must write a one to bit 23 of the Interrupt set-enable register `0x10C`.

——— **Note** ———

If an interrupt is Pending or Active when its enable bit is set to 0, it remains in its current state.

———————————

Interrupts 0-15 fields are read as one, that is, always enabled, and write to these fields have no effect.

Notpresent interrupts (depending on the Interrupt Controller Type Register and interrupt number field) related fields are read as zero and writes to these fields have no effect.

10.5.4 Interrupt Clear-pending and Set-pending registers, 0x200-0x21C and 0x280-0x29C

These registers are used to:

- determine which interrupts are currently in Pending state for at least one MP11 CPU. (An interrupt can be Pending for some MP11 CPUs and Active or Inactive for other MP11 CPUs).
- force some interrupts to enter Pending state by overriding interrupt assertion detection.
- force Pending interrupts to return to Inactive state.

Writing a 1 to a bit in the Clear-pending register means that the corresponding interrupt returns from Pending to Inactive state for all MP11 CPUs. Active state is not modified.

Writing a 1 to a bit in the Set-pending register means that the corresponding interrupt enters Pending state for all MP11 CPUs in the CPU Targets Register.

The values read in Clear-pending and Set-pending registers for the same interrupts range are the same and represent current Pending interrupts. If a bit is read as 1, it implies that the interrupt is Pending for at least one MP11 CPU.

Interrupts 0 to 31 fields are aliased for each MP11 CPU, which means that MP11 CPUs can read different values from these registers. The value read reflects all the Pending interrupts for the accessing MP11 CPU.

All Reserved interrupts, spurious interrupt and notpresent interrupts (depending on Interrupt Controller Type Register and interrupt number field) related fields are read as zero and write to these fields has no effect.

Writing to IT0 to IT15 related fields has no effect. These interrupts can only be triggered through the Software Interrupt Register (see *Software Interrupt Register, 0xF00* on page 10-17).

10.5.5 Active Bit Registers, 0x300-0x31C

The Active Bit Registers are used to determine which interrupts are currently Active (bit read as 1) on at least one MP11 CPU.

Interrupts 0 to 31 fields are aliased for each MP11 CPU, which means that MP11 CPUs can read different values from these registers. The value read reflects all the Active interrupts for the accessing MP11 CPU.

10.5.6 Interrupt priority registers, 0x400-0x4FC

Interrupt priority registers store the priority of an individual interrupt. There can be up to 64 interrupt priority registers. Figure 10-4 shows the format of these registers.

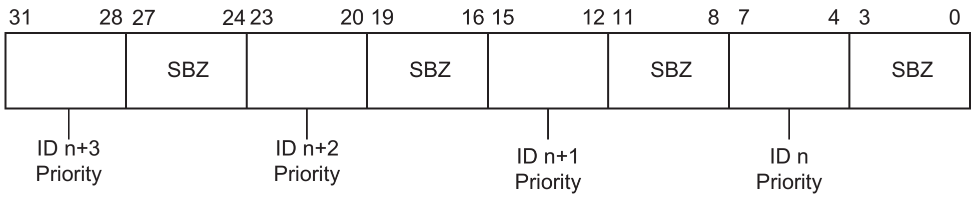


Figure 10-4 Interrupt priority registers format

The first four registers are aliased for each MP11 CPU, that is, the priority set for IT0-15 and IT29-31 can be different for each MP11 CPU. The priority of IPIs IT0-15 depends on the receiving CPU, not the sending CPU.

All Reserved interrupts, spurious interrupts, and notpresent interrupts (depending on the Interrupt Controller Type Register Interrupt number field) related fields are read as zero and writes to these fields have no effect.

Note

Priority zero is the highest priority. The lowest priority is priority `0xF`.

10.5.7 Interrupt CPU targets registers, 0x800-0x8FC

These registers store the list of MP11 CPUs for which an interrupt can be Pending. Each bit in the MP11 CPU target registers refers to one MP11 CPU. For example, value `0x3` means that the interrupt is sent to MP11 CPU 0 and MP11 CPU 1. Interrupt target registers are ignored in cases of software triggered interrupts.

There can be up to 64 interrupt MP11 CPU targets registers. Figure 10-5 shows the format of the interrupt MP11 CPU targets registers.

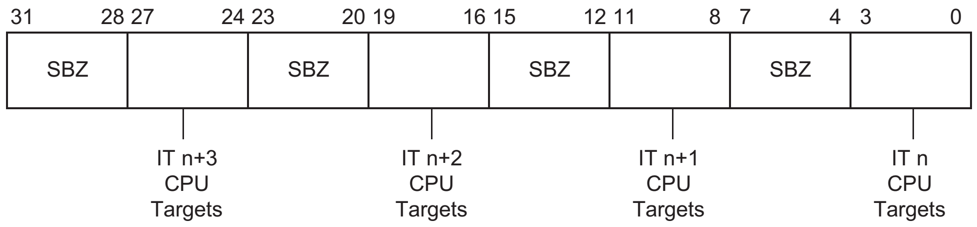


Figure 10-5 Interrupt CPU targets registers format

For IT29, IT30, and IT31, values read in corresponding fields depend on accessing the MP11 CPU because these interrupt sources are private.

- For MP11 CPU 0, CPU targets 29, 30, and 31 are read as `0x1`. Writes are ignored
- For MP11 CPU 1 (if present), CPU targets 29, 30, and 31 are read as `0x2`. Writes are ignored
- For MP11 CPU 2 (if present), CPU targets 29, 30, and 31 are read as `0x4`. Writes are ignored
- For CPU 3 (if present), CPU targets 29, 30, and 31 are read as `0x8`. Writes are ignored

For IT0-IT28, these fields are ignored. They are read as zero and writes are ignored. Targeted CPUs for software triggered interrupts can only be set through the Software Interrupt Registers.

Modifying a CPU target list has no influence on a Pending or Active interrupt, but takes effect on a subsequent assertion of the interrupt.

All Reserved interrupts, spurious interrupts, and notpresent interrupts (depending on the Interrupt Controller Type Register Interrupt number field) related fields are read as zero and writes to these fields have no effect.

10.5.8 Interrupt configuration registers, 0xC00-0xC3C

Interrupt configuration registers defines the assertion condition and the software model of each interrupt. There can be up to 16 interrupt configuration registers. Figure 10-6 shows their format.

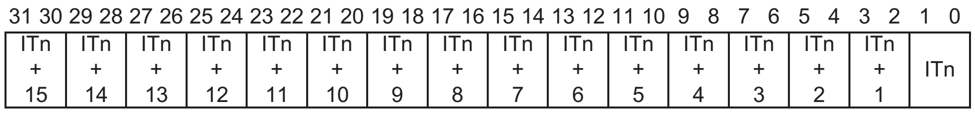


Figure 10-6 Interrupt configuration registers format

Table 10-3 shows the individual ITn encoding for bit 1 and bit 0 of each bit pair.

Table 10-3 Interrupt line encodings for bits 1 and 0

IT bit	Value	Meaning
IT bit [1]	0	The interrupt line is level high active.
	1	The interrupt line is rising edge sensitive.
IT bit [0]	0	The interrupt line uses the N-N software model.
	1	The interrupt line uses the 1-N software model.

All Reserved interrupts, spurious interrupts, and notpresent interrupts (depending on the Interrupt Controller Type Register Interrupt number field) related fields are read as zero and writes to these fields have no effect.

For ID0-ID15, bit 1 of the configuration pair is always read as one, that is, rising edge sensitive.

For ID0-ID15, bit 0 (software model) can be configured and applies to the interrupts sent from the writing MP11 CPU.

For ID29, and ID30, the configuration pair is always read as b10, that is rising edge sensitive and 1-N software model because these IDs are allocated to timer and watchdog interrupts that are CPU-specific.

For ID31, the configuration pair is a Reserved field, and is read as zero, because this ID is allocated to the legacy nIRQ pin which is always level low active whether the Interrupt Controller is active or not.

10.5.9 Software Interrupt Register, 0xF00

The Software Interrupt Register is a write-only register that is used to trigger an interrupt (identified with its ID) to a list of MP11 CPUs. Figure 10-7 shows the Software Interrupt Register format.

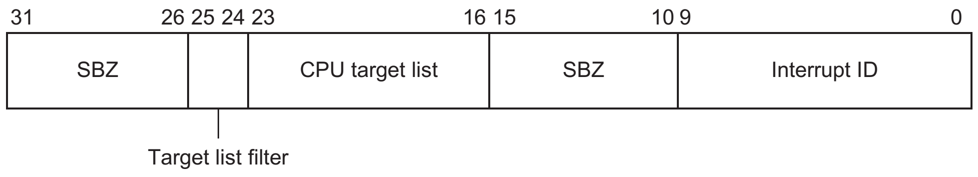


Figure 10-7 Software Interrupt Register format

MP11 CPU target list can be different from the one defined in CPU Targets List Register for the specified interrupt ID. Table 10-4 shows the Software Interrupt Register bit assignments.

Table 10-4 Software Interrupt Register bit assignments

Bit	Value	Meaning
[31:26]	SBZ	
[25:24]	Target list filter	b00 Interrupt sent to the MP11 CPUs listed in CPU Targets List b01 CPU Targets List is ignored, interrupt is sent to all but the requesting CPU. b10 CPU Targets List is ignored, interrupt is sent to the requesting CPU only. b11 RESERVED.
[23:16]	CPU targets list	Notpresent MP11 CPUs in the CPU target list are ignored.
[15:10]	SBZ	-
[9:0]	Interrupt ID	Triggering an interrupt with an ID bigger than the number of supported interrupts has no effect.

10.6 CPU interface registers definition

Each MP11 CPU can access its own CPU Interface at the same address for software coherency through all MP11 CPUs. The SCU is responsible for redirecting MP11 CPU access to the correct MP11 CPU interface depending on its ID.

For debug purposes, individual MP11 CPU interfaces can be addressed by any MP11 CPU because the register range is duplicated in the SCU memory map.

All registers of the MP11 CPU interfaces must be accessible by 32-bit accesses only. Table 10-5 shows the MP11 CPU interface registers.

Table 10-5 MP11 CPU interface registers

Address offset	Type	Reset value	Function	Reference
`0x00`	R/W	0x00000000	Control Register	See *CPU Interface Control Register, 0x00*.
`0x04`	R/W	0x000000F0	Priority Mask Register	See *Priority Mask Register, 0x04* on page 10-19.
`0x08`	R/W	0x00000003	Binary Point Register	See *Binary Point Register, 0x08* on page 10-20.
`0x0C`	RO	0x000003F F	Interrupt Acknowledge Register	See *Interrupt Acknowledge Register, 0x0C* on page 10-20.
`0x10`	WO	-	End of Interrupt Register	See *End of Interrupt (EOI) Register, 0x10* on page 10-21.
`0x14`	RO	0x000000F0	Running Priority Register	See *Running Interrupt Register, 0x14* on page 10-21.
`0x18`	RO	0x000003F F	Highest Pending Interrupt Register	See *Highest Pending Interrupt Register, 0x18* on page 10-22.
Others	-	-	Reserved	-

10.6.1 CPU Interface Control Register, 0x00

Figure 10-8 on page 10-19 shows the format of CPU Interface Control Register.

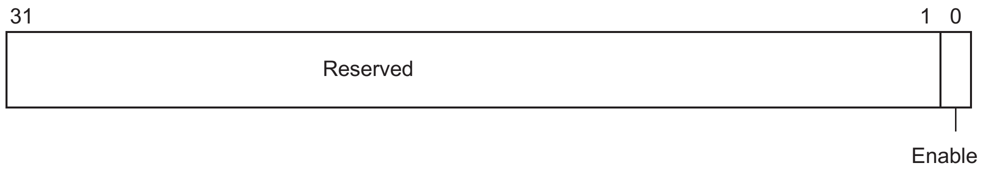


Figure 10-8 CPU Interface Control Register format

Bit 0 is a CPU interface enable. If the bit is 0, no interrupt requests are made to the dedicated MP11 CPU even if there are Pending interrupts for that MP11CPU in the Interrupt Distributor.

Bit 0 also controls the legacy IRQ mode. See *Interrupt Distributor* on page 10-4.

10.6.2 Priority Mask Register, 0x04

The priority mask is used to prevent interrupts from being sent to the MP11 CPU. The CPU Interface asserts an interrupt request to an MP11 CPU if the priority of the highest Pending interrupt sent by the Interrupt Distributor is strictly higher than the mask set in the Priority Mask Register. Figure 10-9 shows the Priority Mask Register bit format.

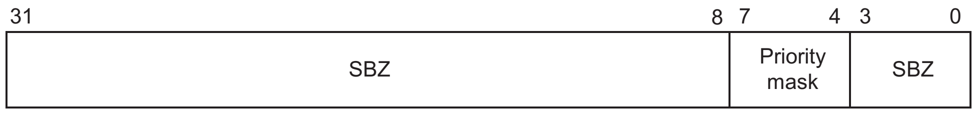


Figure 10-9 Priority Mask Register format

Table 10-6 shows the Priority Mask Register bit assignments.

Table 10-6 Priority Mask Register bit assignments

Bits	Name	Meaning
[31:8]	-	SBZ
[7:4]	Priority Mask	Priority mask values `0xF` Interrupts with priority `0x0-0xE` are not masked. `0x0` All interrupts are masked.
[3:0]	-	SBZ

One consequence of the strict comparison is that a Pending interrupt with the lowest possible priority, `0xF`, never causes the assertion of an interrupt request to MP11 CPUs, permitting an extra level of interrupt enabling.

10.6.3 Binary Point Register, 0x08

The Binary Point Register is used to ignore a certain number of bits in the priority comparison made in the CPU Interface for pre-emption. Figure 10-10 shows the Binary Point Register format.

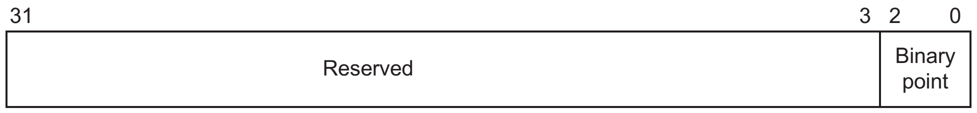


Figure 10-10 Binary Point Register format

Table 10-7 describes the meanings of the binary point bit values.

Table 10-7 Binary point bit values assignment

Bit value	Meaning
b011	All priority bits are compared for pre-emption.
b100	Only bits [7:5] of priority are compared for pre-emption.
b101	Only bits [7:6] of priority are compared for pre-emption.
b110	Only bit [7] of priority is compared for pre-emption.
b111	No pre-emption is performed.

Trying to write a value not listed in Table 10-7 has the same effect as writing b011.

10.6.4 Interrupt Acknowledge Register, 0x0C

The Interrupt Acknowledge Register is a read-only register. The Interrupt Acknowledge Register is used to determine the source of the interrupt request received by an MP11 CPU. If no interrupt is Pending then the Interrupt ID returned is 1023, spurious interrupt. Figure 10-11 on page 10-21 shows the Interrupt Acknowledge Register format.

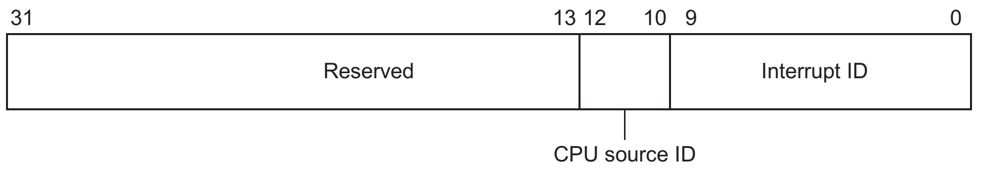


Figure 10-11 Interrupt Acknowledge Register format

Table 10-8 shows the Interrupt Acknowledge Register bit assignments.

Table 10-8 Interrupt Acknowledge Register bit assignments

Bit	Name	Function
[31:13]	-	SBZ/RAZ
[12:10]	CPU Source ID	CPU source ID depends on Interrupt ID field: If the Interrupt ID field is 0-15, it contains the ID of the CPU that requested the IPI. In all other cases, the CPU source ID field is read as zero and can be ignored.
[9:0]	Interrupt ID	Interrupt Identifier

10.6.5 End of Interrupt (EOI) Register, 0x10

This write-only register is used when the software finishes handling an interrupt. The End of Interrupt Register has the same format as the Interrupt Acknowledge Register shown in Figure 10-11.

10.6.6 Running Interrupt Register, 0x14

This read-only register contains the priority level of the last acknowledged and not completed interrupt on this MP11CPU. Figure 10-12 shows the format of this register.

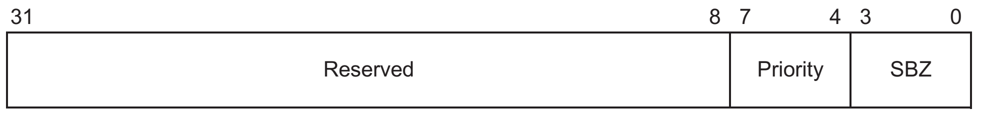


Figure 10-12 Running Interrupt Register format

10.6.7 Highest Pending Interrupt Register, 0x18

The Highest Pending Interrupt Register contains the Interrupt ID (and CPU source ID as appropriate) of the highest pending interrupt being presented to the CPU Interface by the Interrupt Distributor. If no interrupt is Pending then the Interrupt ID returned is 1023, spurious interrupt.

The format of this register is the same as the Interrupt Acknowledge Register shown in Figure 10-11 on page 10-21.

10.7 Timer and Watchdog blocks

There are timer and watchdog blocks for each MP11 CPU present in the ARM11 MPCore processor. Both timer and watchdog blocks have the following features:

- A 32-bit counter that generates an interrupt when it reaches zero.
- An 8-bit prescaler to enable better control of the period.
- Configurable single-shot or auto-reload modes.
- Configurable starting values for the counter.

The watchdog can be configured as a timer.

10.7.1 Calculating timer intervals

The timer interval is calculated using the following equation:

$$\left(\frac{(\text{PRESCALER\_value}+1) \times (\text{Load\_value}+1) \times 2}{\text{CPU CLK\_frequency}}\right)$$

This equation can be used to calculate the period between two events out of the timers and the watchdog time-out time.

10.7.2 Timer and watchdog registers

Addresses are given relative to the base address of the region for the Interrupt Distributor defined by the SCU memory map (see *About the MPCore private memory region* on page 9-2). All timer and watchdog registers are word accessible only. Table 10-9 shows the timer and watchdog registers. All registers not described in Table 10-9 are Reserved.

Table 10-9 Timer and watchdog registers

Offset	Type	Reset Value	Name
`0x00`	R/W	`0x00000000`	Timer Load Register
`0x04`	R/W	`0x00000000`	Timer Counter Register
`0x08`	R/W	`0x00000000`	Timer Control Register
`0x0C`	R/W	`0x00000000`	Timer Interrupt Status Register
`0x20`	R/W	`0x00000000`	Watchdog Load Register
`0x24`	R/W	`0x00000000`	Watchdog Counter Register
`0x28`	R/W	`0x00000000`	Watchdog Control Register

Table 10-9 Timer and watchdog registers

Offset	Type	Reset Value	Name
`0x2C`	R/W	`0x00000000`	Watchdog Interrupt Status Register
`0x30`	R/W	`0x00000000`	Watchdog Reset Sent Register
`0x34`	WO	-	Watchdog Disable Register

10.7.3 Timer Load Register, 0x00

The Timer Load Register is a 32-bit register that contains the value copied to the Timer Counter Register when it decrements down to zero with auto reload mode enabled. Writing to the Timer Load Register means that you also write to the Timer Counter Register.

10.7.4 Timer Counter Register, 0x04

The Timer Counter Register is a down counter.

The Timer Counter Register decrements if the timer is enabled using the timer enable bit in the Timer Control Register. If the MP11 CPU belonging to the timer is in debug state, the counter does not decrement until the MP11 CPU returns to non debug state.

When the Timer Counter Register reaches zero and auto reload mode is enabled, it reloads the value in the Timer Load Register and then decrements from that value. If auto reload mode is not enabled the Timer Counter Register decrements down to zero and stops.

When the Timer Counter Register reaches zero, the timer interrupt status event flag is set and the interrupt ID 29 is set as Pending in the Interrupt Distributor, if interrupt generation is enabled in the Timer Control Register.

Writing to the Timer Counter Register or Timer Load Register forces the Timer Counter Register to decrement from the newly written value.

10.7.5 Timer Control Register, 0x08

Figure 10-13 on page 10-25 shows the Timer Control Register format.

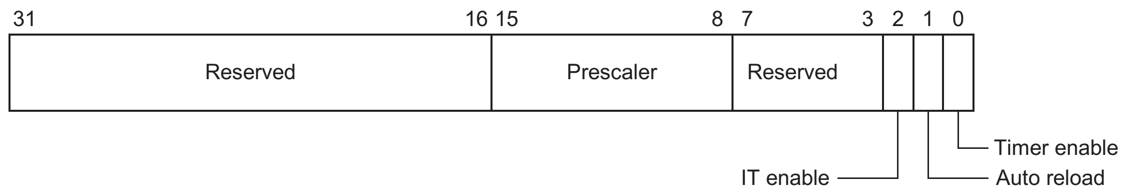


Figure 10-13 Timer Control Register format

Table 10-10 shows the Timer Control Register bit assignments.

Table 10-10 Timer Control Register bit assignments

Bit	Name	Function
[31:16]	-	SBZ/RAZ
[15:8]	Prescaler	The prescaler modifies the clock period for the decrementing event for the Counter Register. See *Calculating timer intervals* on page 10-23 for the equation.
[7:3]	-	SBZ/RAZ
[2]	IT Enable	If set, the interrupt ID 29 is set as Pending in the Interrupt Distributor when the event flag is set in the Timer Status Register.
[1]	Auto-reload	1'b0 Single shot mode. Counter decrements down to zero, sets the event flag and stops. 1'b1 Auto-reload mode. Each time the Counter Register reaches zero, it is reloaded with the value contained in the Load Register and then continues decrementing.
[0]	Timer Enable	Global timer enable 1'b0 Timer is disabled and the counter does not decrement. All registers can still be read or/and written 1'b1 Timer is enabled and the counter decrements normally.

10.7.6 Timer Interrupt Status Register, 0x0C

Figure 10-14 on page 10-26 shows the Timer Interrupt Status Register format.

The event flag is a sticky bit that is automatically set when the Counter Register reaches zero. If the timer interrupt is enabled, Interrupt ID 29 is set as Pending in the Interrupt Distributor after the event flag is set.

The event flag is cleared when written to 1. Trying to write a zero to the event flag or a one when it is not set has no effect.

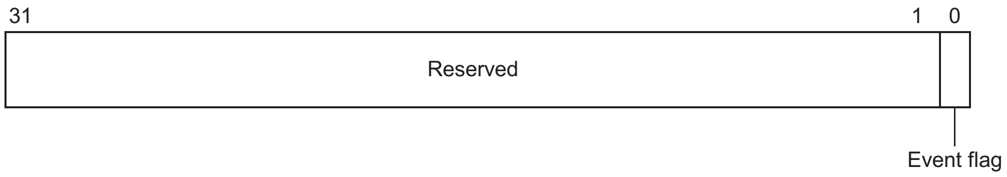


Figure 10-14 Timer Interrupt Status Register format

10.7.7 Watchdog Load Register, 0x20

The Watchdog Load Register is a 32-bit register that contains the value copied to the Watchdog Counter Register when it decrements down to zero with auto reload mode enabled, in Timer mode. Writing to the Watchdog Load Register means that you also write to the Watchdog Counter Register.

10.7.8 Watchdog Counter Register, 0x24

The Watchdog Counter Register is a down counter.

It decrements if the Watchdog is enabled using the Watchdog enable bit in the Watchdog Control Register. If the MP11 CPU belonging to the Watchdog is in debug state, the counter does not decrement until the MP11 CPU returns to non debug state.

When the Watchdog Counter Register reaches zero and auto reload mode is enabled, and in timer mode, it reloads the value in the Watchdog Load Register and then decrements from that value. If auto reload mode is not enabled or the watchdog is not in timer mode, the Watchdog Counter Register decrements down to zero and stops.

When in watchdog mode the only way to update the Watchdog Counter Register is to write to the Watchdog Load Register. When in timer mode the Watchdog Counter Register is write accessible.

The behavior of the watchdog when the Watchdog Counter Register reaches zero depends on its current mode:

Timer mode When the Watchdog Counter Register reaches zero, the watchdog interrupt status event flag is set and the interrupt ID 30 is set as Pending in the Interrupt Distributor, if interrupt generation is enabled in the Watchdog Control Register.

Watchdog mode

If a software failure prevents the Watchdog Counter Register from being refreshed, the Watchdog Counter Register reaches zero, the Watchdog reset status flag is set and the associated **RESETREQ** reset request output pin is asserted. The external reset source is then responsible for resetting all or part of the ARM11 MPCore processor.

10.7.9 Watchdog Control Register, 0x28

Figure 10-15 shows the Watchdog Control Register format.

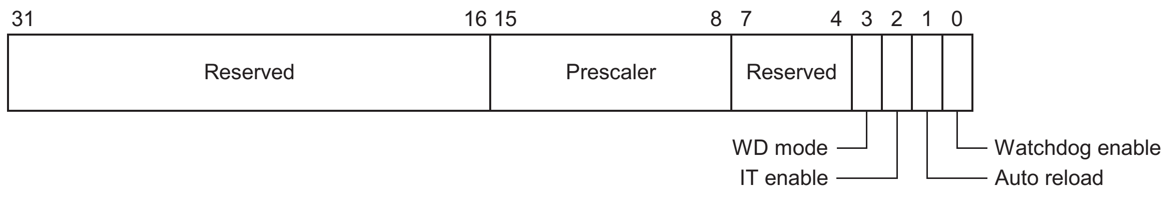


Figure 10-15 Watchdog Control Register format

Table 10-11 shows the Watchdog Control Register bit assignments.

Table 10-11 Watchdog Control Register bit assignments

Bit	Name	Function
[31:16]	-	SBZ/RAZ
[15:8]	Prescaler	The prescaler modifies the clock period for the decrementing event for the Counter Register. See *Calculating timer intervals* on page 10-23.
[7:4]	-	SBZ/RAZ
[3]	Watchdog mode	1'b0 Timer mode, default Writing a zero to this bit has no effect. You must use the Watchdog Disable Register to put the watchdog into timer mode. 1'b1 Watchdog mode.

Table 10-11 Watchdog Control Register bit assignments (continued)

Bit	Name	Function
[2]	IT Enable	If set, the interrupt ID 30 is set as Pending in the Interrupt Distributor when the event flag is set in the watchdog Status Register. In watchdog mode this bit is ignored.
[1]	Auto-reload	1'b0 Single shot mode. Counter decrements down to zero, sets the event flag and stops. 1'b1 Auto-reload mode. Each time the Counter Register reaches zero, it is reloaded with the value contained in the Load Register and then continues decrementing. In watchdog mode this bit is ignored.
[0]	Watchdog Enable	Global watchdog enable 1'b0 Watchdog is disabled and the counter does not decrement. All registers can still be read and /or written 1'b1 Watchdog is enabled and the counter decrements normally.

10.7.10 Watchdog Interrupt Status Register, 0x2C

Figure 10-16 shows the Watchdog Interrupt Status Register format.

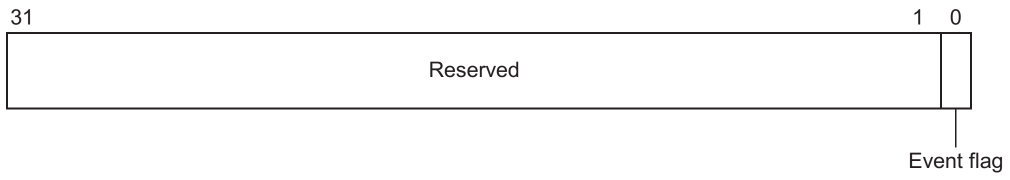


Figure 10-16 Watchdog Interrupt Status Register format

The event flag is a sticky bit that is automatically set when the Counter Register reaches zero in timer mode. If the watchdog interrupt is enabled, Interrupt ID 30 is set as Pending in the Interrupt Distributor after the event flag is set.

The event flag is cleared when written to 1. Trying to write a zero to the event flag or a one when it is not set has no effect.

10.7.11 Watchdog Reset Status Register, 0x30

Figure 10-17 shows the Watchdog Reset Status Register format.

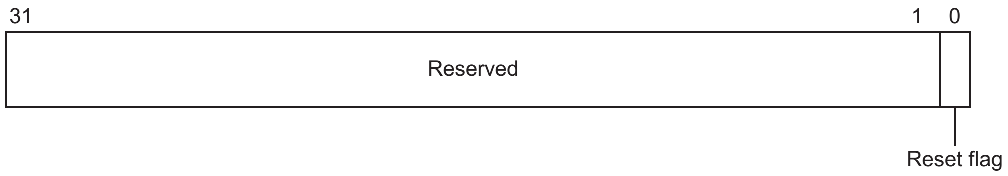


Figure 10-17 Watchdog Reset Status Register format

The reset flag is a sticky bit that is automatically set when the Counter Register reaches zero and a reset request is sent accordingly. (In watchdog mode)

The reset flag is cleared when written to 1. Trying to write a zero to the reset flag or a one when it is not set has no effect.

This flag is not reset by normal CPU resets but has its own reset line that must not be asserted when the CPU reset assertion is the result of a watchdog reset request. This distinction enables software to differentiate between a normal boot sequence, reset flag is zero, and one caused by a previous watchdog time-out, reset flag set to one.

10.7.12 Watchdog Disable Register, 0x34

The Watchdog Disable Register is a 32-bit write-only register used to switch from watchdog to timer mode. The software must write `0x12345678` then `0x87654321` successively to the Watchdog Disable Register so that the watchdog mode bit in the Watchdog Control Register is set to zero.

If one of the values written to the Watchdog Disable Register is incorrect or if any other write occurs in between the two word writes, the watchdog remains in its current state. To reactivate the Watchdog, the software must write 1 to the watchdog mode bit of the Control Register.

Chapter 11
Clocking, Resets, and Power Management

This chapter describes the clocking and reset options available for ARM11 MPCore processors. It contains the following sections:

11.1 Clocking

The ARM11 MPCore processor has one functional clock input, CLK. All four MP11 CPUs, and SCUs are clocked with a distributed version of CLK. The Distributed Interrupt Controller, and private timers and watchdog are clocked with a divided version of CLK. **ic_clk** is generated by enabling CLK one cycle out of two.

11.1.1 Synchronous clocking

The ARM11 MPCore processor does not have any asynchronous interfaces. So, all the bus interfaces and the interrupt signals must be synchronous with reference to CLK. The fact that the Distributed Interrupt Controller is clocked at half the clock frequency implies that all the interrupt signals entering the ARM11 MPCore processor must last at least two clock cycles.

The AXI bus clock domain can be run at n:1 (AXI:Core) ratio to CLK using the **ACLKEN** signal.

11.2 Reset

The ARM11 MPCore processor has the following reset inputs:

nSCURESET	The **nSCURESET** signal is the main processor reset that initializes the majority of the ARM11 MPCore processor logic, except MP11 CPU logic.
nCPURESET[3:0]	The **nCPURESET[3:0]** signals are the main CPU resets that initialize the majority of the MP11 CPU logic, except the CP14 debug logic. There is one reset per MP11 CPU.
nWDRESET[3:0]	The **nWDRESET[3:0]** signals are used to reset the watchdog reset status flag. These resets must be asserted for power-on reset but not if one MPCore reset assertion is caused by a watchdog reset request.
DBGnTRST	The **DBGnTRST** signal is the DBGTAP reset.
nPORESET	The **nPORESET**signal is the power-on reset that initializes the CP14 debug logic. See *CP14 debug instructions* on page 12-24 for details.

All of these are active LOW signals that reset logic in the ARM11 MPCore processor. You must take care when designing the logic to drive these reset signals.

RESETREQ[3:0] output pins can be used to trigger the reset assertion for one or multiple MP11 CPUs. A RESETREQ is asserted when a private MP11 CPU watchdog counter decrements down to zero (when in watchdog mode), signalling a potential software error.

11.3 Reset modes

The reset signals present in the ARM11 MPCore processor design enable you to reset different parts of the design independently. Table 11-1 shows the reset signals, and the combinations and possible applications that you can use them in.

Table 11-1 Reset modes

Mode	nTRST	nSCURESET	nPORESET[3:0]	nWDRESET[3:0]	nCPURESET[3:0]
Power-on reset All CPUs reset and SCU and interrupt distribution	x	0	All 0	All 0	All 0
Individual power-on reset, out of shutdown or dormant mode All CPU logic (including debug) reset	x	1	[n] = 0	[n] = 0	[n] = 0
Soft reset CPU logic reset (excluding debug)	x	1	All 1	[n] = 0	[n] = 0
Reset from Watchdog request CPU logic reset excluding debug and Watchdog reset flag.	x	1	All 1	All 1	[n] = 0
Reset DBGTAP logic embedded in MPCore (TDI/TMS/TCK/RTCK generation and synchronization)	0	x	x	x	x
No Reset. Normal run mode.	x	1	1	1	1

Note

For any combination not in Table 11-1, the behavior is architecturally Unpredictable.

11.3.1 Power-on reset

You must apply power-on or *cold* reset to the ARM11 MPCore processor when power is first applied to the system. In the case of power-on reset, the leading (falling) edge of the reset signals, **nSCURESET, nCPURESET[3:0]** and **nWDRESET[3:0]**, does not have to be synchronous to **CLK but the rising edge must be**.

It is recommended that you assert the reset signals for at least three **CLK** cycles to ensure correct reset behavior. Adopting a three-cycle reset eases the integration of other ARM parts into the system.

It is not necessary to assert **DBGnTRST** on power-up.

11.3.2 CP14 debug logic

Because the **nPORESET** signal is synchronized within the ARM11 MPCore processor, you do not have to synchronize this signal.

11.3.3 MP11 CPU reset

A processor or *warm* reset initializes the majority of the MP11 CPU, excluding the MP11 CPU DBGTAP controller and the EmbeddedICE-RT logic. Processor reset is typically used for resetting a system that has been operating for some time, for example, watchdog reset.

11.3.4 ARM11 MPCore processor reset

It is strongly recommended that if one MP11 CPU has to be reset, the entire ARM11 MPCore processor is reset. That is, all present MP11 CPUs, SCUs, and Interrupt Controllers must be reset to prevent any potential system deadlock caused by an MP11 CPU being reset while it was performing a transaction on the memory system.

11.3.5 DBGTAP reset

DBGTAP reset initializes the state of the MP11 CPU DBGTAP controller. DBGTAP reset is typically used by the RealView™ ICE module for hot connection of a debugger to a system.

DBGTAP reset enables initialization of the DBGTAP controller without affecting the normal operation of the ARM11 MPCore processor.

11.3.6 Normal operation

During normal operation, neither processor reset nor power-on reset is asserted. If the DBGTAP port is not being used, the value of **DBGnTRST** does not matter.

11.4 About Power Consumption Control

The features of the ARM11 MPCore processor that improve energy efficiency include:

- accurate branch and return prediction, reducing the number of incorrect instruction fetch and decode operations
- use of physically addressed caches, which reduces the number of cache flushes and refills, saving energy in the system
- the use of MicroTLBs reduces the power consumed in translation and protection lookups each cycle
- the caches use sequential access information to reduce the number of accesses to the Tag RAMs and to unwanted Data RAMs.

In the ARM11 MPCore processor extensive use is also made of gated clocks and gates to disable inputs to unused functional blocks. Only the logic actively in use to perform a calculation consumes any dynamic power.

11.5 Individual MP11 CPU Power Control

Place holders for level-shifters and clamps are inserted around each MP11 CPU so that implementation of different power domains can be eased. It is the responsibility of software to signal to the Snoop Control Unit and the Distributed Interrupt Controller that an individual MP11 CPU will be shut-off so that the MP11 CPU can be seen as non-existent in the cluster.

The debug scan chain of such non-visible MP11 CPUs must be bypassed so that all other powered CPUs can still be accessed.

Each MP11 CPU can be in one of the following modes:

Run mode everything is clocked and powered-up

WFI mode CPU clock is stopped. Only logic needed for wake-up is still active.

Dormant mode
everything is powered off except RAM arrays that are in retention mode.

Powered-off everything is powered-off.

Table 11-2 shows the individual power modes.

Table 11-2 MP11 CPU power modes

Mode	MP11 CPU logic	RAM arrays	Wake-up mechanism
Run Mode	Powered-up Everything clocked	Powered-up	N/A
WFI/WFE	Powered-up Only wake-up logic clocked	Powered-up	Wake-up on interrupts (external or timer/WD). L1 memory system only wake-up in case of SCU coherency request.
Dormant	Powered-off	Retention state/voltage	External wake-up event to power controller.
Powered- off	Powered-off	Powered-off	External wake-up event to power controller.

Entry to Dormant or powered-off mode must be controlled through an external power controller. The CPU Status Register in the SCU is used in conjunction with CPU WFI entry flag to signal to the power controller which power domain it can cut, using the PWRCTL bus (see *SCU CPU Status Register* on page 9-6).

11.5.1 Run mode

Run mode is the normal mode of operation in which all of the functionality of the core is available.

11.5.2 Wait for interrupt (WFI/WFE) mode

Wait for Interrupt mode disables most of the clocks of a CPU, while keeping its logic powered up. This reduces the power drawn to the static leakage current, plus a tiny clock power overhead required to enable the device to wake up from the WFI state. The transition from the WFI mode to the Run mode is caused by:

- an interrupt, masked or unmasked
- a debug request, regardless of whether debug is enabled
- a reset.

The debug request can be generated by an externally generated debug request, using the **EDBGRQ** pin on the ARM11 MPCore processor, or from a Debug Halt instruction issued to the MP11 CPU through the debug scan chains.

Entry into WFI Mode is performed by executing the Wait For Interrupt instruction. To ensure that the memory system is not affected by the entry into the Standby state, the following operations are performed:

- A Data Synchronization Barrier, so ensuring that all explicit memory accesses occurring in program order before the WFI have completed. This avoids any possible deadlocks that could be caused in a system where memory access will trigger or enable an interrupt that the core is waiting for.
- Any other memory accesses that have been started at the time that the WFI instruction is executed will be completed as normal. This ensures that the Level 2 memory system does not see any disruption caused by the WFI.
- The debug channel remains active throughout a WFI.

11.5.3 Shutdown mode

Shutdown mode has the entire device powered down, and all state, including cache, must be saved externally by software. The part is returned to the run state by the assertion of reset. This state saving is performed with interrupts disabled, and finishes with a DrainWriteBuffer operation. The MP11 CPU then communicates with a power controller that the device is ready to be powered down in the same manner as when entering Dormant Mode.

11.5.4 Dormant mode

Dormant mode is designed to allow the core to be powered down, while leaving the caches powered up and maintaining their state.

The RAM blocks that are to remain powered up must be implemented on a separate power domain, and there is a need to clamp all of the inputs to the RAMs to a known logic level (with the chip enable being held inactive). This clamping is not implemented in gates as part of the default synthesis flow because it would contribute to a tight critical path. Implementations which wish to implement Dormant mode must add these clamps around the RAMs, either as explicit gates in the RAM power domain, or as pull-down transistors which clamp the values while the core is powered down.

The RAM blocks that must remain powered up during Dormant mode are:

- all Data RAMs associated with the cache
- all Tag RAMs associated with the cache
- all Dirty RAMs associated with the cache.

Before entering Dormant mode, the state of the MP11 CPU, excluding the contents of the RAMs that remain powered up in dormant mode, must be saved to external memory. These state saving operations must ensure that the following occur:

- All ARM registers, including CPSR and SPSR registers are saved.
- All CP15 registers are saved.
- All debug-related state must be saved.
- MP11 CPU must correctly set the CPU Status Register in the SCU so that it enters Dormant Mode (see TBD).
- A Data Synchronization Barrier instruction is executed to ensure that all state saving has been completed.
- The MP11 CPU then communicates with the power controller that it is ready to enter dormant mode by performing a WFI instruction so that power control output reflects the value of SCU CPU Status Register (see *SCU CPU Status Register* on page 9-6).
- On entry into Dormant mode, the Reset signal to the ARM11 MPCore processor must be asserted by the external power control mechanism.

Transition from Dormant state to Run state is triggered by the external power controller. The external power controller must assert reset to the MP11 CPU until the power is restored. After power is restored, the core leaves reset, and by interrogating the power control register in SCU, can determine that the saved state must be restored.

11.5.5 Communication to the Power Management Controller

Communication between the ARM11 MPCore processor and the external Power Management Controller can be performed using the **PWRCTLOn** MPCore output signals and MPCore input clamp signals.

PWRCTLOn MPCore output signals

These signals constrain the external Power Management Controller. The value of PWRCTLOn depends on the value of the SCU CPU Status Register (see *SCU CPU Status Register* on page 9-6). The SCU CPU Status Register value is only copied on **PWRCTLOn** after the CPU signals that it is ready to entry low power mode by executing a WFI instruction and subsequent **STANDBYWFI** pin assertion.

MPCore input signals

BISTCLAMP, **DEBUGCLAMP**, **CPUCLAMP[3:0]**, and **RAMCLAMP[4:0]** are used by the external Power Management Controller to isolate MPCore power domains from one another before they are turned off. These signals are only meaningful if the ARM11 MPCore processor has been implemented with level shifters and power domain clamps designed in.

11.6 IEM Support

The IEM infrastructure is intended to be supported at the system level to allow the end user to choose at which level in the SOC to separate different power domains.

There are placeholders between MPCore logic and RAM arrays so that implementation of level-shifters for these parts can be on a different power domain.

11.6.1 MPCore power domains

The ARM11 MPCore processor has up to ten power domains:

- four power domains for MP11 CPUs logic cells
- four power domains for MP11 CPUs caches and TLB RAMs
- one power domain for SCU TAG RAMs
- one power domain for remaining logic, the SCU logic cells, and private peripherals.

Figure 11-1 on page 11-12 shows all the power domains and where placeholders are inserted for power domain isolation.

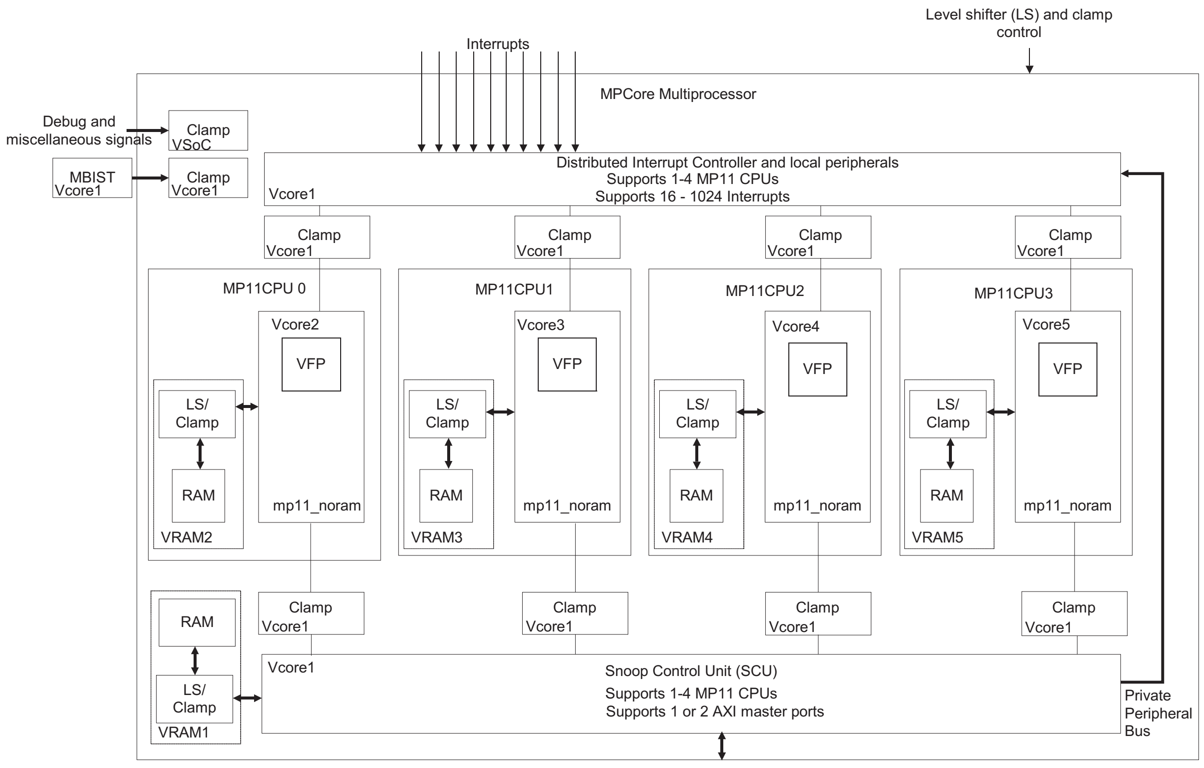


Figure 11-1 ARM11 MPCore processor power management

11.7 Debug

All MP11 CPUs are independent from a debug point of view. They can each be independently put into debug state or independently started from debug state.

DBGACK and **DBGREQ** signals are provided for each MP11 CPU so that external logic can be added to synchronize MP11 CPUs debug state entry. This can be useful to help debug of SMP software running on multiple MP11 CPUs.

A possible debug synchronization policy is:

- Whenever one MP11 CPU running in SMP mode enters debug, **DBGACK** HIGH, all the remaining MP11 CPUs running in SMP mode receive an external debug request through their DBGREQ pin. This ensures that the SMP cores enter debug at almost the same time, on the boundary of two instructions.
- To exit from debug state, the Restart instruction must be scanned into all stopped MP11 CPUs. When the state machines enter Run-Test/Idle state, normal operations resume. When Run-Test/Idle state is entered, all the processors resume operation simultaneously.

11.7.1 Debug and power management

Figure 11-2 on page 11-14 shows how the TAP controllers can be daisy-chained together for a two MP11 CPU solution.

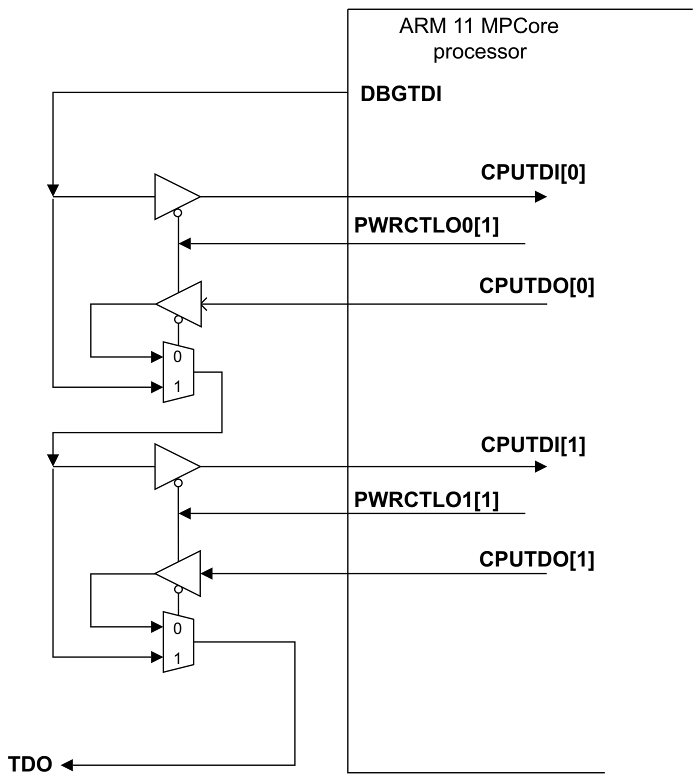


Figure 11-2 Daisy-chaining debug signals

The buffers represent clamps that allow separating different power domains in case of power-down of some cores.

Chapter 12
Debug

This chapter contains details of the MPCore debug unit to assist the development of application software, operating systems, and hardware. It contains the following sections:

12.1 Debug systems

The ARM11 MPCore processor forms one component of a debug system that interfaces from the high-level debugging performed by you, to the low-level interface supported by the ARM11 MPCore processor. Figure 12-1 shows a typical system.

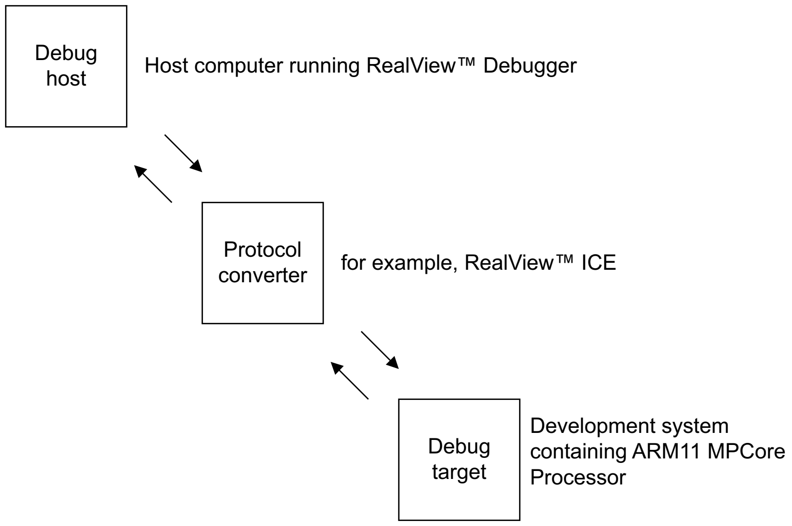

Debug host
Host computer running RealView™ Debugger
Protocol converter
for example, RealView™ ICE
Debug target
Development system containing ARM11 MPCore Processor

Figure 12-1 Typical debug system

This typical system has three parts:

- *The debug host*
- *The protocol converter*
- *The ARM11 MPCore processor* on page 12-3.

12.1.1 The debug host

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

12.1.2 The protocol converter

The debug host is connected to the MPCore development system using an interface, for example an RS232. The messages broadcast over this connection must be converted to the interface signals of the ARM11 MPCore processor. This function is performed by a protocol converter, for example, RealView ICE.

See Appendix D *Scan chain ordering with RVI* for information on RealView ICE and scan chain ordering.

12.1.3 The ARM11 MPCore processor

The ARM11 MPCore processor, with debug unit, is the lowest level of the system. The debug extensions enable you to:

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

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

12.2 About the debug unit

The MPCore debug unit assists in debugging software running on the ARM11 MPCore processor. You can use an MPCore debug unit, in combination with a software debugger program, to debug:

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

The debug unit enables you to:

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

you can debug the ARM11 MPCore processor in the following ways:

- *Halt mode debugging*
- *Monitor debug-mode debugging*.

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

12.2.1 Halt mode debugging

When the MPCore debug unit is in Halt mode, the processor halts when a debug event, such as a breakpoint, occurs. When the core is halted, an external host can examine and modify its state using the DBGTAP.

In Halt mode you can examine and alter all processor state (processor registers), coprocessor state, memory, and input/output locations through the DBGTAP. This mode is intentionally invasive to program execution. Halt mode requires:

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

See *CP14 c1, Debug Status and Control Register (DSCR)* on page 12-9 to learn how to set the MPCore debug unit into Halt mode.

12.2.2 Monitor debug-mode debugging

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

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

When debugging in Monitor debug-mode, you can program new debug events through CP14. This coprocessor is the software interface of all the debug resources such as the breakpoint and watchpoint registers. See *CP14 c1, Debug Status and Control Register (DSCR)* on page 12-9 to learn how to set the MPCore debug unit into Monitor debug-mode.

12.2.3 Virtual Addresses and debug

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

The terms *Instruction Virtual Address* (IVA) and *Data Virtual Address* (DVA), where used, mean the VA corresponding to an instruction address and the VA corresponding to a data address respectively.

12.2.4 Programming the debug unit

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

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

CP14 is accessed using coprocessor instructions in Monitor debug-mode, and certain debug scan chains in Halt mode, see Chapter 13 *Debug Test Access Port* to learn how to access the MPCore debug unit using scan chains.

12.3 Debug registers

Table 12-1 shows definitions of terms used in register descriptions.

Table 12-1 Terms used in register descriptions

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

On a power-on reset, all the CP14 debug registers take the values indicated by the Reset value column in the register bit field definition tables (Table 12-4 on page 12-10, Table 12-6 on page 12-15, Table 12-9 on page 12-17, Table 12-11 on page 12-20, and Table 12-12 on page 12-21). In these tables, - means an Undefined reset value.

12.3.1 Accessing debug registers

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

Table 12-2 on page 12-7 shows the CP14 debug register map. All of these registers are also accessible as scan chains from the DBGTAP.

Table 12-2 CP14 debug register map

Binary address		Register number	CP14 debug register name	Abbreviation
Opcode_2	CRm			
b000	b0000	c0	Debug ID Register	DIDR
b000	b0001	c1	Debug Status and Control Register	DSCR
b000	bb0010-b0100	c2-c4	Reserved	-
b000	b0101	c5	Data Transfer Register	DTR
b000	b0110	c6	Reserved	-
b000	b0111	c7	Vector Catch Register	VCR
b000	b1000-b1111	c8-c15	Reserved	-
b001-b011	b0000-b1111	c16-c63	Reserved	-
b100	b0000-b0101	c64-c69	Breakpoint Value Registers	BVRy[a]
	b0110-b111	c70-c79	Reserved	-
b101	b0000-b0101	c80-c85	Breakpoint Control Registers	BCRy[a]
	b0110-b1111	c86-c95	Reserved	-
b110	b0000-b0001	c96-c97	Watchpoint Value Registers	WVRy[a]
	b0010-b1111	c98-c111	Reserved	-
b111	b0000-b0001	c112-c113	Watchpoint Control Registers	WCRy[a]
	b0010-b1111	c114-c127	Reserved	-

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

Note

All the debug resources required for Monitor debug-mode debugging are accessible through CP14 registers. For Halt mode debugging some additional resources are required. See Chapter 13 *Debug Test Access Port*.

12.3.2 CP14 c0, Debug ID Register (DIDR)

The Debug ID Register is a read-only register that defines the configuration of debug registers in a system. Figure 12-2 shows the Debug ID Register format.

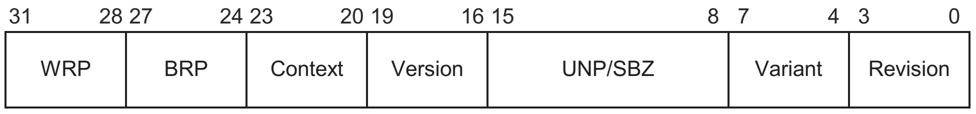


Figure 12-2 Debug ID Register format

The MPCore r0p1 processor has `0x15110001` in this register.

Table 12-3 shows the bit field definitions for the Debug ID Register.

Table 12-3 Debug ID Register bit field definition

Bits	Type	Description

[31:28] WRP	RO	Number of Watchpoint Register Pairs:
		b0000 = 1 WRP
		b0001 = 2 WRPs
		…
		b1111 = 16 WRPs.
		For the ARM11 MPCore processor these bits are b0001 (2 WRPs).

[27: 24] BRP	RO	Number of Breakpoint Register Pairs: b0000 = Reserved. The minimum number of BRPs is 2. b0001 = 2 BRPs b0010 = 3 BRPs … b1111 = 16 BRPs. For the ARM11 MPCore processor these bits are b0101 (6 BRPs).

[23: 20] Context	RO	Number of Breakpoint Register Pairs with context ID comparison capability:
		b0000 = 1 BRP has context ID comparison capability
		b0001 = 2 BRPs have context ID comparison capability
		…
		b1111 = 16 BRPs have context ID comparison capability.
		For the ARM11 MPCore processor these bits are b0001 (2 BRPs).

[19:16] Version	RO	Debug architecture version.

Table 12-3 Debug ID Register bit field definition (continued)

Bits	Type	Description
[15:8]	UNP/SBZP	Reserved.
[7: 4] Variant	RO	Implementation-defined variant number. This number is incremented on functional changes.
[3: 0] Revision	RO	Implementation-defined revision number. This number is incremented on bug fixes.

The values of the following fields of the Debug ID Register agree with the values in CP15 c0, ID Register:

- DIDR[3:0] is the same as CP15 c0 bits [3:0]
- DIDR[7:4] is the same as CP15 c0 bits [23:20].

See *c0, ID Code Register* on page 3-12 for a description of CP15 c0, ID Register.

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

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

The Debug Status and Control Register contains status and configuration information about the state of the debug system. Figure 12-3 shows the format of the Debug Status and Control Register.

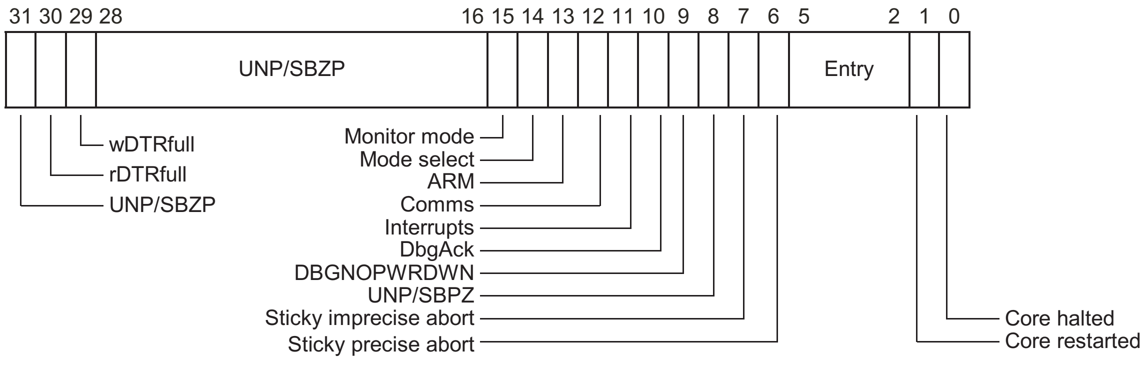


Figure 12-3 Debug Status and Control Register format

Figure 12-4 on page 12-10 shows the relationship between the core restarted bit and the core halted bit.

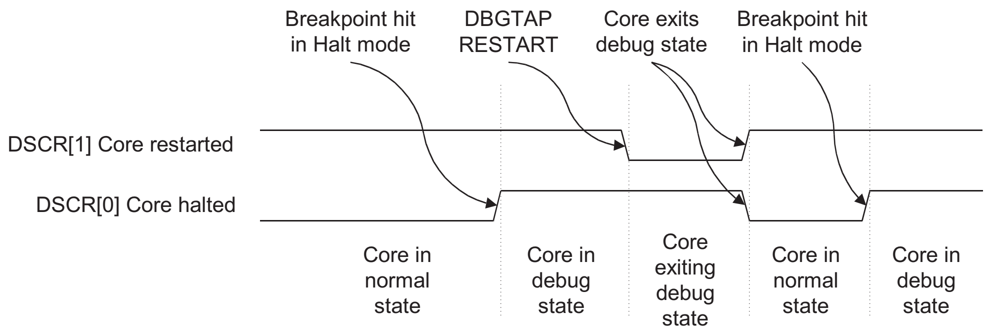


Figure 12-4 Core restarted bit and core halted bit

Table 12-4 shows the bit field definitions for the Debug Status and Control Register.

Table 12-4 Debug Status and Control Register bit field definitions

Bits	Core view	External view	Reset value	Description
[31]	UNP/SBZP	UNP/SBZP	-	Reserved.
[30]	R	R	0	The rDTRfull flag: 0 = rDTR empty 1 = rDTR full. This flag is automatically set on writes by the DBGTAP debugger to the rDTR and is cleared on reads by the core of the same register. No writes to the rDTR are enabled if the rDTRfull flag is set.
[29]	R	R	0	The wDTRfull flag: 0 = wDTR empty 1 = wDTR full. This flag is automatically cleared on reads by the DBGTAP debugger of the wDTR and is set on writes by the core to the same register.
[28:16]	UNP/SBZP	UNP/SBZP	-	Reserved.
[15]	RW	R	0	The Monitor debug-mode enable bit: 0 = Monitor debug-mode disabled 1 = Monitor debug-mode enabled. For the core to take a debug exception, Monitor debug-mode must be both selected and enabled (bit 14 clear and bit 15 set).

Table 12-4 Debug Status and Control Register bit field definitions (continued)

Bits	Core view	External view	Reset value	Description
[14]	R	RW	0	Mode select bit: 0 = Monitor debug-mode selected 1 = Halt mode selected and enabled.
[13]	R	RW	0	Execute ARM instruction enable bit: 0 = Disabled 1 = Enabled. If this bit is set, the core can be forced to execute ARM instructions in debug state using the Debug Test Access Port. If this bit is set when the core is not in debug state, the behavior of the ARM11 MPCore processor is Unpredictable.
[12]	RW	R	0	User mode access to comms channel control bit: 0 = User mode access to comms channel enabled 1 = User mode access to comms channel disabled. If this bit is set and a User mode process tries to access the DIDR, DSCR, or the DTR, the Undefined instruction exception is taken. Because accessing the rest of CP14 debug registers is never possible in User mode (see *Executing CP14 debug instructions* on page 12-25, setting this bit means that a User mode process cannot access any CP14 debug register.
[11]	R	RW	0	Interrupts bit: 0 = Interrupts enabled 1 = Interrupts disabled. If this bit is set, the IRQ and FIQ input signals are inhibited.[a]
[10]	R	RW	0	DbgAck bit. If this bit is set, the **DBGACK** output signal (see *External signals* on page 12-47) is forced HIGH, regardless of the processor state.[a]
[9]	R	RW	0	Powerdown disable: 0 = **DBGNOPWRDWN** is LOW 1 = **DBGNOPWRDWN** is HIGH. See *External signals* on page 12-47.
[8]	UNP/SBZP	UNP/SBZP	-	Reserved.

Table 12-4 Debug Status and Control Register bit field definitions (continued)

Bits	Core view	External view	Reset value	Description
[7]	R	RC	0	Sticky imprecise Data Aborts bit: 0 = No imprecise Data Aborts occurred since the last time this bit was cleared 1 = An imprecise Data Abort has occurred since the last time this bit was cleared. It is cleared on reads of a DBGTAP debugger to the DSCR.
[6]	R	RC	0	Sticky precise Data Abort bit: 0 = No precise Data Abort occurred since the last time this bit was cleared 1 = An precise Data Abort has occurred since the last time this bit was cleared. This flag is meant to detect Data Aborts generated by instructions issued to the processor using the Debug Test Access Port. Therefore, if the DSCR[13] execute ARM instruction enable bit is a 0, the value of the sticky precise Data Abort bit is Unpredictable. It is cleared on reads of a DBGTAP debugger to the DSCR.

[5:2]	RW	R	b0000	Method of entry bits:
				b0000 = a Halt DBGTAP instruction occurred
				b0001 = a breakpoint occurred
				b0010 = a watchpoint occurred
				b0011 = a BKPT instruction occurred
				b0100 = an **EDBGRQ** signal activation occurred
				b0101 = a vector catch occurred
				b0110 = a data-side abort occurred
				b0111 = an instruction-side abort occurred
				b1xxx = reserved.

[1]	R	R	1	Core restarted bit: 0 = the processor is exiting debug state 1 = the processor has exited debug state. After executing a DBGTAP IR instruction, the debugger polls this bit until it is set to 1 so it knows that the IR instruction took effect. Polling DSCR[0] until it is set to 0 is not safe because the processor could exit debug state and re-enter it (because of other debug event) before the debugger samples the DSCR.[b] See *Debug state* on page 12-33 for a definition of debug state.

Table 12-4 Debug Status and Control Register bit field definitions (continued)

Bits	Core view	External view	Reset value	Description
[0]	R	R	0	Core halted bit: 0 = the processor is in normal state 1 = the processor is in debug state. After programming a debug event, the debugger polls this bit until it is set to 1 so it knows that the processor entered debug state. See *Debug state* on page 12-33 for a definition of debug state.

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

b. See Figure 12-4 on page 12-10 for the relationship between the core restarted and halted bits.

Bits [5:2] are set to indicate:

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

Using bits [5:2], a Prefetch Abort or a Data Abort handler determines if it must jump to the monitor target. Additionally, a DBGTAP debugger or monitor target can determine the specific debug event that caused the debug state or debug exception entry.

12.3.4 CP14 c5, Data Transfer Registers (DTR)

This register consists of two separate physical registers:

- the rDTR (Read Data Transfer Register)
- the wDTR (Write Data Transfer Register).

The register accessed is dependent on the instruction used:

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

Note

Read and write refer to the core view.

For details of the use of these registers with the rDTRfull flag and wDTRfull flag see *Debug communications channel* on page 12-37. Figure 12-5 on page 12-14 shows the format of both the rDTR and wDTR.

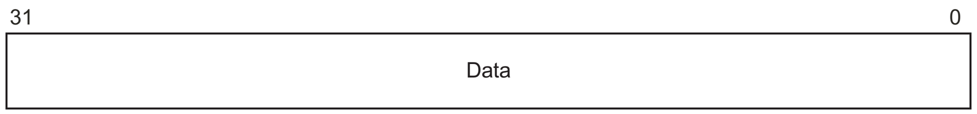


Figure 12-5 DTR format

Table 12-5 shows the bit field definitions for rDTR and wDTR.

Table 12-5 Data Transfer Register bit field definitions

Bits	Core view	External view	Description
[31:0]	RO	WO	Read data transfer register (read-only)
[31:0]	WO	RO	Write data transfer register (write-only)

12.3.5 CP14 c7, Vector Catch Register (VCR)

The ARM11 MPCore processor supports efficient exception vector catching which is controlled by the VCR. Figure 12-6 shows the Vector Catch Register format.

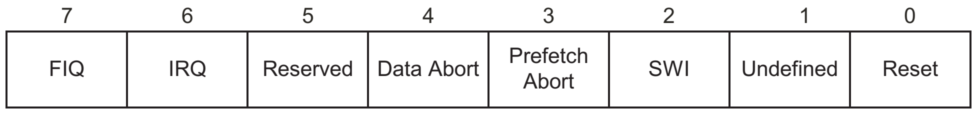


Figure 12-6 Vector Catch Register format

If one of the bits in this register is set and the corresponding vector is committed for execution, then a Debug exception or debug state entry might be generated, depending on the value of the DSCR[15:14] bits (see *Behavior of the processor on debug events* on page 12-28). Under this model, any kind of fetch of an exception vector can trigger a vector catch, not just the ones due to exception entries.

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

Bits [31:8] and bit 5 are reserved.

TBD Table 12-6 shows the bit field definitions for the Vector Catch Register.

Table 12-6 Vector Catch Register bit field definitions

Bits	Attributes	Reset value	Description	Normal address	High vector address
[31:8]	UNP/SBZP	-	Reserved	-	-
[7]	RW	0	Vector catch enable, FIQ	`0x0000001C`	`0xFFFF001C`
[6]	RW	0	Vector catch enable, IRQ	`0x00000008`	`0xFFFF0008`
[5]	UNP/SBZP	-	Reserved	-	-
[4]	RW	0	Vector catch enable, Data Abort	`0x00000010`	`0xFFFF0010`
[3]	RW	0	Vector catch enable, Prefetch Abort	`0x0000000C`	`0xFFFF000C`
[2]	RW	0	Vector catch enable, SWI	`0x00000008`	`0xFFFF0008`
[1]	RW	0	Vector catch enable, Undefined Instruction	`0x00000004`	`0xFFFF0004`
[0]	RW	0	Vector catch enable, Reset	`0x00000000`	`0xFFFF0000`

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

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

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

The BVR of a BRP is loaded with an IVA and then its contents can be compared against the IVA bus of the processor.

The breakpoint value contained in the BVR corresponds to either an IVA or a context ID. Breakpoints can be set on:

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

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

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

Table 12-7 shows the breakpoint and watchpoint registers that are implemented in the ARM11 MPCore processor.

Table 12-7 MPCore breakpoint and watchpoint registers

Binary address		Register number	CP14 debug register name	Abbreviation	Context ID capable?
Opcode_2	CRm				
b100	b0000-b0011	c64-c67	Breakpoint Value Registers 0-3	BVR0-3	No
	b0100-b0101	c68-c69	Breakpoint Value Registers 4-5	BVR4-5	Yes
	b0110-b1111	c70-c79	Reserved	-	-
b101	b0000-b0011	c80-c83	Breakpoint Control Registers 0-3	BCR0-3	No
	b0100-b0101	c84-c85	Breakpoint Control Registers 4-5	BCR4-5	Yes
	b0110-b1111	c86-c95	Reserved	-	-
b110	b0000-b0001	c96-c97	Watchpoint Value Registers 0-1	WVR0-1	-
	b0010-b1111	c98-c111	Reserved	-	-
b111	b0000-b0001	c112-c113	Watchpoint Control Registers 0-1	WCR0-1	-
	b0010-b1111	c114-c127	Reserved	-	-

Table 12-8 shows the bit field definitions for the context ID and the non context ID Breakpoint Value Registers.

Table 12-8 Breakpoint Value Registers bit field definition

Context ID capable?	Bits	Attributes	Description
No	[31:2]	RW	Breakpoint address
Yes	[31:0]	RW	Breakpoint address

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

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

These registers contain the necessary control bits for setting:

- breakpoints
- linked breakpoints.

Figure 12-7 shows the format of the Breakpoint Control Registers.

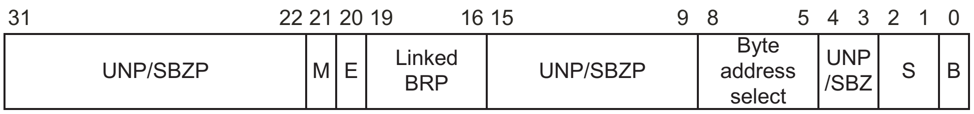


Figure 12-7 Breakpoint Control Registers, format

Table 12-9 shows the bit field definitions for the Breakpoint Control Registers.

Table 12-9 Breakpoint Control Registers, bit field definitions

Bits	Attributes	Reset value	Description
[31:22]	UNP/SBZP	-	Reserved.
[21]	RW (Read as 0)	- (-)	0 = Instruction Virtual Address. The corresponding BVR is compared against the IVA bus. 1 = Context ID. The corresponding BVR is compared against the CP15 context ID (register 13). If this BRP does not have context ID comparison capability, this control bit does not apply and the corresponding bit is read as 0. See Table 12-10 on page 12-19 for details.
[20]	RW	-	Enable linking: 0 = Linking disabled 1 = Linking enabled. When this bit is set HIGH, the corresponding BRP is linked. See Table 12-10 on page 12-19 for details.
[19:16]	RW	-	Linked BRP number. The binary number encoded here indicates another BRP to link this one with. If a BRP is linked with itself, it is Unpredictable if a breakpoint debug event is generated.
[15:9]	UNP/SBZP	-	Reserved.

Table 12-9 Breakpoint Control Registers, bit field definitions (continued)

Bits	Attributes	Reset value	Description
[8:5]	RW	-	Byte address select. The BVR is programmed with a word address. You can use this field to program the breakpoint so it hits only if certain byte addresses are accessed. b0000 = The breakpoint never hits bxxx1= If the byte at address BVR[31:2]+0 is accessed, the breakpoint hits bxx1x = If the byte at address BVR[31:2]+1 is accessed, the breakpoint hits bx1xx = If the byte at address BVR[31:2]+2 is accessed, the breakpoint hits b1xxx = If the byte at address BVR[31:2]+3 is accessed, the breakpoint hits. This field must be set to b1111 when this BRP is programmed for context ID comparison, that is BCR[21:20] set to b1x. Otherwise breakpoint or watchpoint debug events might not be generated as expected. **Note** These are little-endian byte addresses. This ensures that a breakpoint is triggered regardless of the endianness of the instruction fetch. For example, if a breakpoint is set on a certain Thumb instruction by doing BCR[8:5] = b0011, it is triggered if in little-endian and IVA[1:0] is b00 or if big-endian and IVA[1:0] is b10.
[4:3]	UNP/SBZP	-	Reserved

[2:1]	RW	-	b00 = Reserved
			b01= Privileged
			b10 = User
			b11 = Either.
			If this BRP is programmed for context ID comparison and linking (BCR[21:20] is set b11), then the BCR[2:1] field of the IVA-holding BRP takes precedence and it is Undefined whether this field is included in the comparison or not. Therefore, it must be set to either.
			The WCR[2:1] field of a WRP linked with this BRP also takes precedence over this field.

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

Table 12-10 summarizes the meaning of BCR bits [21:20].

Table 12-10 Meaning of BCR[21:20] bits

BCR[21:20]	Meaning
b00	The corresponding BVR is compared against the IVA bus. This BRP is not linked with any other one. It generates a breakpoint debug event on an IVA match.
b01	The corresponding BVR is compared against the IVA bus. This BRP is linked with the one indicated by BCR[19:16] linked BRP field. They generate a breakpoint debug event on a joint IVA and context ID match.
b10	The corresponding BVR is compared against CP15 Context Id Register, c13. This BRP is not linked with any other one. It generates a breakpoint debug event on a context ID match.
b11	The corresponding BVR is compared against CP15 Context Id Register, c13. Another BRP (of the BCR[21:20]=b01 type), or WRP (with WCR[20]=b1), is linked with this BRP. They generate a breakpoint or watchpoint debug event on a joint IVA or DVA and context ID match.

Note

The BCR[8:5] and BCR[2:1] fields still apply when a BRP is set for context ID comparison. See *Setting breakpoints, watchpoints, and vector catch debug events* on page 12-39 for detailed programming sequences for linked breakpoints and linked watchpoints.

The following rules apply to the ARM11 MPCore processor for breakpoint debug event generation:

- The update of a BVR or a BCR can take effect several instructions after the corresponding MCR. It takes effect by the next IMB.
- Updates of the CP15 Context ID Register c13, can take effect several instructions after the corresponding MCR. However, the write takes place by the end of the exception return. This is to ensure that a User mode process, switched in by a processor scheduler, can break at its first instruction.
- Any BRP (holding an IVA) can be linked with any other one with context ID capability. Several BRPs (holding IVAs) can be linked with the same context ID capable one.

- If a BRP (holding an IVA) is linked with one that is not configured for context ID comparison and linking, it is Unpredictable whether a breakpoint debug event is generated or not. BCR[21:20] fields of the second BRP must be set to b11.
- If a BRP (holding an IVA) is linked with one that is not implemented, it is Unpredictable if a breakpoint debug event is generated or not.
- If a BRP is linked with itself, it is Unpredictable if a breakpoint debug event is generated or not.
- If a BRP (holding an IVA) is linked with another BRP (holding a context ID value), and they are not both enabled (both BCR[0] bits set), the first one does not generate any breakpoint debug event.

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

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

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

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

- a DVA
- a DVA/context ID pair.

For the second case a WRP and a BRP with context ID comparison capability have to be linked. A debug event is generated when both the DVA and the context ID pair match simultaneously. Table 12-11 shows the bit field definitions for the Watchpoint Value Registers.

Table 12-11 Watchpoint Value Registers, bit field definitions

Bits	Attributes	Reset value	Description
[31:2]	RW	-	Watchpoint address

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

These registers contain the necessary control bits for setting:

- watchpoints
- linked watchpoints.

Figure 12-8 on page 12-21 shows the format of the Watchpoint Control Registers.

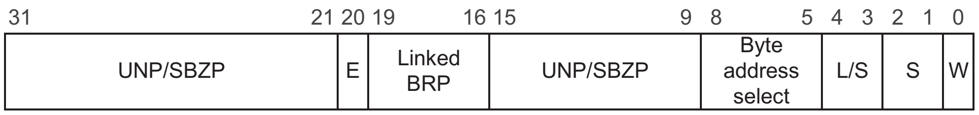


Figure 12-8 Watchpoint Control Registers, format

Table 12-12 shows the bit field definitions for the Watchpoint Control Registers.

Table 12-12 Watchpoint Control Registers, bit field definitions

Bits	Attributes	Reset value	Description
[31:21]	UNP/SBZP	-	Reserved.
[20]	RW	-	Enable linking bit: 0 = Linking disabled 1 = Linking enabled. When this bit is set, this watchpoint is linked with the context ID holding BRP selected by the linked BRP field.
[19:16]	RW	-	Linked BRP. The binary number encoded here indicates a context ID holding BRP to link this WRP with.
[15:9]	SBZ	-	Reserved.
[8:5]	RW	-	Byte address select. The WVR is programmed with a word address. This field can be used to program the watchpoint so it hits only if certain byte addresses are accessed. b0000 = The watchpoint never hits bxxx1= If the byte at address WVR[31:2]+0 is accessed, the watchpoint hits bxx1x = If the byte at address WVR[31:2]+1 is accessed, the watchpoint hits bx1xx = If the byte at address WVR[31:2]+2 is accessed, the watchpoint hits b1xxx = If the byte at address WVR[31:2]+3 is accessed, the watchpoint hits. **Note** These are little-endian byte addresses. This ensures that a watchpoint is triggered regardless of the way it is accessed. For example, if a watchpoint is set on a certain byte in memory by doing WCR[8:5] = b0001. `LDRB r0, #0x0` it triggers the watchpoint in little-endian mode, as does `LDRB r0, #x3` in legacy big-endian mode (B bit of CP15 c1 set).

Table 12-12 Watchpoint Control Registers, bit field definitions (continued)

Bits	Attributes	Reset value	Description

[4:3]	RW	-	Load or store access. The watchpoint can be conditioned to the type of access being done:
			b00 = Reserved
			b01 = Load
			b10 = Store
			b11 = Either.
			A SWP triggers on Load, Store, or Either. A load exclusive instruction, LDREX, triggers on Load or Either. A store exclusive instruction, STREX, triggers on Store or Either, whether it succeeded or not.

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

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

In addition to the rules for breakpoint debug event generation, see *CP14 c80-c85, Breakpoint Control Registers (BCR)* on page 12-17, the following rules apply to the ARM11 MPCore processor for watchpoint debug event generation:

- The update of a WVR or a WCR can take effect several instructions after the corresponding MCR. It is only guaranteed to have taken effect by the next 1MB.
- Any WRP can be linked with any BRP with context ID comparison capability. Several BRPs (holding IVAs) and WRPs can be linked with the same context ID capable BRP.
- If a WRP is linked with a BRP that is not configured for context ID comparison and linking, it is Unpredictable if a watchpoint debug event is generated or not. BCR[21:20] fields of the BRP must be set to b11.
- If a WRP is linked with a BRP that is not implemented, it is Unpredictable if a watchpoint debug event is generated or not.
- If a WRP is linked with a BRP and they are not both enabled (BCR[0] and WCR[0] set), it does not generate a watchpoint debug event.

12.4 CP14 registers reset

nPORESET[0], nPORESET[1], nPORESET[2],or nPORESET[3] reset all the CP14 debug registers for MP11 CPU0, MP11 CPU1, MP11 CPU2, or MP11 CPU3 respectively (see *Power-on reset* on page 11-5).

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

12.5 CP14 debug instructions

Table 12-13 shows the CP14 debug instructions.

Table 12-13 CP14 debug instructions

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

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

b000	b0111	7	VCR	`MRC p14, 0, <Rd>, c0, c7, 0`[a] `MCR p14, 0, <Rd>, c0, c7, 0`[a]
b100	b0000-b1111	64-79	BVR	`MRC p14, 0, <Rd>, c0, cy,4`[ab] `MCR p14, 0, <Rd>, c0, cy,4`[ab]
b101	b0000-b1111	80-95	BCR	`MRC p14, 0, <Rd>, c0, cy,5`[ab] `MCR p14, 0, <Rd>, c0, cy,5`[ab]
b110	b0000-b1111	96-111	WVR	`MRC p14, 0, <Rd>, 0, cy, 6`[ab] `MCR p14, 0, <Rd>, 0, cy, 6`[ab]
b111	b0000-b1111	112-127	WCR	`MRC p14, 0, <Rd>, c0, cy, 7`[ab] `MCR p14, 0, <Rd>, c0, cy, 7`[ab]

a. <Rd> is any of R0-14 ARM registers.
b. y is the decimal representation for the binary number CRm.

In Table 12-13, `MRC p14,0,<Rd>,c0,c5,0` and `STC p14,c5,<addressing mode>` refer to the rDTR and `MCR p14,0,<Rd>,c0,c5,0` and `LDC p14,c5,<addressing mode>` refer to the wDTR. See *CP14 c5, Data Transfer Registers (DTR)* on page 12-13 for more details.

The `MRC p14,0,R15,c0,c1,0` instruction sets the CPSR flags as follows:

- N flag = DSCR[31]. This is an Unpredictable value.
- Z flag = DSCR[30]. This is the value of the rDTRfull flag.

- C flag = DSCR[29]. This is the value of the wDTRfull flag.
- V flag = DSCR[28]. This is an Unpredictable value.

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

12.5.1 Executing CP14 debug instructions

If the core is in debug state (see *Debug state* on page 12-33), you can execute any CP14 debug instruction regardless of the processor mode.

If the processor tries to execute a CP14 debug instruction that either is not in Table 12-13 on page 12-24, or is targeted to a reserved register, such as a non-implemented BVR, the Undefined instruction exception is taken.

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

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

When DSCR bit 14 is set (Halt mode selected and enabled), if the software running on the processor tries to access any register other than the DIDR, the DSCR, or the DTR, the core takes the Undefined instruction exception. The same thing happens if the core is not in any debug mode (DSCR[15:14]=b00).

This lockout mechanism ensures that the software running on the core cannot modify the settings of a debug event programmed by the DBGTAP debugger.

Table 12-14 on page 12-26 shows the results of executing CP14 debug instructions.

Table 12-14 Debug instruction execution

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

12.6 Debug events

A debug event is any of the following:

- *Software debug event*
- *External debug request signal* on page 12-28
- *Halt DBGTAP instruction* on page 12-28.

12.6.1 Software debug event

A software debug event is any of the following:

- A watchpoint debug event. This occurs when:
    - — the DVA present in the data bus matches the watchpoint value
    - — all the conditions of the WCR match
    - — the watchpoint is enabled
    - — the linked contextID-holding BRP (if any) is enabled and its value matches the context ID in CP15 c13.
- A breakpoint debug event. This occurs when:
    - — an instruction was fetched and the IVA present in the instruction bus matched the breakpoint value
    - — at the same time the instruction was fetched, all the conditions of the BCR matched
    - — the breakpoint was enabled
    - — at the same time the instruction was fetched, the linked contextID-holding BRP (if any) was enabled and its value matched the context ID in CP15 c13
    - — the instruction is now committed for execution.
- A breakpoint debug event also occurs when:
    - — an instruction was fetched and the CP15 Context ID (register 13) matched the breakpoint value
    - — at the same time the instruction was fetched, all the conditions of the BCR matched
    - — the breakpoint was enabled
    - — the instruction is now committed for execution.
- A software breakpoint debug event. This occurs when a BKPT instruction is committed for execution.

- A vector catch debug event. This occurs when:
  - The instruction at a vector location was fetched. This includes any kind of prefetches, not just the ones due to exception entry.
  - At the same time the instruction was fetched, the corresponding bit of the VCR was set (vector catch enabled).
  - The instruction is now committed for execution.

12.6.2 External debug request signal

The ARM11 MPCore processor has an external debug request input signal, **EDBGRQ**. When this signal is HIGH it causes the processor to enter debug state when execution of the current instruction has completed. When this happens, the DSCR[5:2] method of entry bits are set to b0100.

EDBGRQ must stay HIGH until DBGACK is asserted.

12.6.3 Halt DBGTAP instruction

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

12.6.4 Behavior of the processor on debug events

This section describes how the processor behaves on debug events while not in debug state. See *Debug state* on page 12-33 for information on how the processor behaves while in debug state.

When a software debug event occurs and Monitor debug-mode is selected and enabled then a Debug exception is taken. However, Prefetch Abort and Data Abort Vector catch debug events are ignored. This is to avoid the processor ending in an unrecoverable state on certain combinations of exceptions and vector catches. Unlinked context ID breakpoint debug events are also ignored if the processor is running in a privileged mode and Monitor debug-mode is selected and enabled.

The external debug request signal and the Halt DBGTAP instruction are ignored when Monitor debug-mode is selected and enabled.

When a debug event occurs and Halt mode is selected and enabled then the processor enters debug state.

When neither Halt nor Monitor debug-mode is selected and enabled, all debug events are ignored, although the BKPT instruction generates a Prefetch Abort exception.

Table 12-15 Behavior of the processor on debug events

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

a. When debug is disabled, a BKPT instruction generates a Prefetch Abort exception instead of being ignored.

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

12.6.5 Effect of a debug event on CP15 registers

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

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

They are set under the following circumstances:

- The IFSR is set whenever a breakpoint, software breakpoint, or vector catch debug event generates a Debug exception entry. It is set to indicate the cause for the Prefetch Abort vector fetch.
- The DFSR is set whenever a watchpoint debug event generates a Debug exception entry. It is set to indicate the cause for the Data Abort vector fetch.
- The ARM11 MPCore processor sets the FAR to an Unpredictable value.
- The WFAR is set whenever a watchpoint debug event generates either a Debug exception or debug state entry. It is set to the VA of the instruction that caused the Watchpoint debug event, plus an offset dependent on the processor state. Table 12-18 on page 12-35 shows the offsets that are used.

Table 12-16 shows the setting of CP15 registers on debug events.

Table 12-16 Setting of CP15 registers on debug events

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

You must take care when setting a breakpoint or software breakpoint debug event inside the Prefetch Abort or Data Abort exception handlers, or when setting a watchpoint debug event on a data address that might be accessed by any of these handlers. These debug events overwrite the r14_abt, SPRS_abt and the CP15 registers listed in this section, leading to an unpredictable software behavior if the handlers did not have the chance of saving the registers.

12.7 Debug exception

When a Software debug event occurs and Monitor debug-mode is selected and enabled then a Debug exception is taken. Prefetch Abort and Data Abort Vector catch debug events are ignored though. Unlinked context ID breakpoint debug events are also ignored if the processor is running in a privileged mode and Monitor debug-mode is selected and enabled.

If the cause of the Debug exception is a watchpoint debug event, the processor performs the following actions:

- The DSCR[5:2] method of entry bits are set to indicate that a watchpoint occurred.
- The CP15 DFSR, FAR, and WFAR, are set as described in *Effect of a debug event on CP15 registers* on page 12-29.
- The same sequence of actions as in a Data Abort exception is performed. This includes setting the r14_abt, base register and destination registers to the same values as if this was a Data Abort.

The Data Abort handler is responsible for checking the DFSR or DSCR[5:2] bit to determine if the routine entry was caused by a debug exception or a Data Abort exception. On entry:

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

Note

- The FAR is set to an Unpredictable value
- The address of the instruction that caused the watchpoint debug event can be found in the WFAR
- The address of the instruction to restart at plus `0x08` can be found in the r14_abt register.

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

- the DSCR[5:2] method of entry bits are set appropriately

- the CP15 IFSR register is set as described in *Effect of a debug event on CP15 registers* on page 12-29.
- the same sequence of actions as in a Prefetch Abort exception is performed.

The Prefetch Abort handler is responsible for checking the IFSR or DSCR[5:2] bits to find out if the routine entry is caused by a Debug exception or a Prefetch Abort exception. If the cause is a Debug exception it branches to the monitor target.

Note

The address of the instruction causing the Software debug event plus `0x04` can be found in the r14_abt register.

Table 12-17 shows the values in the link register after exceptions.

Table 12-17 Values in the link register after exceptions

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

a. Watchpoints can be imprecise.
RA is not the address of the instruction just after the one that hit the watchpoint. The processor might stop a number of instructions later. The address of the instruction that hit the watchpoint is in the CP15 WFAR.

12.8 Debug state

When the conditions in *Behavior of the processor on debug events* on page 12-28 are met then the processor switches to debug state. While in debug state, the processor behaves as follows:

- The DSCR[0] core halted bit is set.
- The **DBGACK** signal is asserted, see *External signals* on page 12-47.
- The DSCR[5:2] method of entry bits are set appropriately.
- The CP15 IFSR, DFSR, and FAR registers are set as described in *Effect of a debug event on CP15 registers* on page 12-29. The WFAR is set to an Unpredictable value.
- The processor is halted. The pipeline is flushed and no instructions are fetched.
- The processor does not change the execution mode. The CPSR is not altered.
- Interrupts and exceptions are treated as described in *Interrupts* on page 12-35 and *Exceptions* on page 12-35.
- Software debug events are ignored.
- The external debug request signal is ignored.
- Debug state entry request commands are ignored.
- There is a mechanism, using the Debug Test Access Port, where the core is forced to execute an ARM state instruction. This mechanism is enabled using DSCR[13] execute ARM instruction enable bit.
- The core executes the instruction as if it is in ARM state, regardless of the actual value of the T and J bits of the CPSR. If you do set both the J and T bits the behavior is Unpredictable.
- In this state the core can execute any ARM state instruction, as if in a privileged mode. For example, if the processor is in User mode then the MSR instruction updates the PSRs and all the CP14 debug instructions can be executed. However, the processor still accesses the register bank and memory as indicated by the CPSR mode bits. For example, if the processor is in User mode then it sees the User mode register bank, and accesses the memory without any privilege.
- The PC behaves as described in *Behavior of the PC in debug state* on page 12-34.

- A DBGTAP debugger can force the processor out of debug state by issuing a Restart instruction, see Table 13-1 on page 13-6. The Restart command clears the DSCR[1] core restarted flag. When the processor has actually exited debug state, the DSCR[1] core restarted bit is set and the DSCR[0] core halted bit and **DBGACK** signal are cleared.

12.8.1 Behavior of the PC in debug state

In debug state:

- The PC is frozen on entry to debug state. That is, it does not increment on the execution of ARM instructions. However, branches and instructions that modify the PC directly do update it.
- If the PC is read after the processor has entered debug state, it returns a value as described in Table 12-18 on page 12-35, depending on the previous state and the type of debug event.
- If a sequence for writing a certain value to the PC is executed while in debug state, and then the processor is forced to restart, execution starts at the address corresponding to the written value. However, the CPSR must be set to the return ARM, Thumb, or Java state before the PC is written to, otherwise the processor behavior is Unpredictable.
- If the processor is forced to restart without having performed a write to the PC, the restart address is Unpredictable.
- If the PC or CPSR are written to while in debug state, subsequent reads to the PC return an Unpredictable value.
- If a conditional branch is executed and it fails its condition code, an Unpredictable value is written to the PC.
- If you switch the processor from ARM to Java state while in debug state, R5[9:0] is cleared. So, to avoid losing any processor state in this situation, save R5 before switching from ARM to Java and restore it afterwards.

Table 12-18 shows the read PC value after debug state entry for different debug events.

Table 12-18 Read PC value after debug state entry

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

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

12.8.2 Interrupts

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

12.8.3 Exceptions

Exceptions are handled as follows while in debug state:

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

Prefetch Abort

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

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

SWI

If this instruction is executed while in debug state, the behavior of the ARM11 MPCore processor is Unpredictable.

Undefined instruction exceptions

If an Undefined instruction is executed while the processor is in Java and debug state, the behavior of the ARM11 MPCore processor is Unpredictable. If an Undefined instruction is executed while the processor is in ARM debug state or Thumb debug state, the behavior of the core is as follows:

- the PC, CPSR, and SPSR_und are set as for normal processor state exception entry
- R14_und is set to an Unpredictable value
- the processor remains in debug state and does not fetch the exception vector.

Data abort

When a Data Abort occurs in debug state, the behavior of the core is as follows:

- The PC, CPSR, and SPSR_abt are set as for a normal processor state exception entry.
- If the debugger has not written to the PC or the CPSR while in debug state, R14_abt is set as described in the *ARM Architecture Reference Manual*.
- If the debugger has written to the PC or the CPSR while in debug state, R14_abt is set to an Unpredictable value.
- The processor remains in debug state and does not fetch the exception vector.
- The DFSR, and FAR are set as for a normal processor state exception entry. The WFAR is set to an Unpredictable value.
- The DSCR[6] sticky precise Data Abort bit, or the DSCR[7] sticky imprecise Data Aborts bit are set.
- The DSCR[5:2] method of entry bits are set to b0110.

If it is an imprecise Data Abort and the debugger has not written to the PC or CPSR, R14_abt is set as described in the *Architecture Reference Manual*. Therefore the processor is in the same state as if the exception was taken on the instruction that was cancelled by the debug state entry sequence. This is necessary because it is not possible to guarantee that the debugger reads the PC before an imprecise Data Abort exception is taken.

12.9 Debug communications channel

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

- The debug communications channel, when the core is not in debug state. It is defined as the set of resources used for communicating between the DBGTAP debugger and a piece of software running on the core.
- The mechanism for forcing the core to execute ARM instructions, when the core is in debug state. For details see *Executing instructions in debug state* on page 13-23.

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

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

The DBGTAP debugger side of the debug communications channel is described in *Monitor debug-mode debugging* on page 13-49.

12.10 Debugging in a system with TLBs

Debugging in a system with TLBs must be as non-intrusive as possible. In MP11 CPUs there is a way to put the main TLB in a state where its contents are not affected by the debugging process. This facility must be accessible from both the core and the DBGTAP debugger side. The ARM11 MPCore processor enables you to put the main TLB in this mode using CP15 c15. See *TLB Debug Control Register* on page 3-72.

12.11 Monitor debug-mode debugging

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

For situations that can only tolerate a small intrusion into the instruction stream, Monitor debug-mode is ideal. Using this technique, code can be suspended with an exception long enough to save off state information and important variables. The code continues when the exception handler is finished. The method of entry bits in the DSCR can be read to determine what caused the exception.

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

12.11.1 Entering the monitor target

Monitor debug-mode is the default mode on power-on reset. Only a DBGTAP debugger can change the mode bit in the DSCR. When a software debug event occurs (as described in *Software debug event* on page 12-27) and Monitor debug-mode is selected and enabled, then a Debug exception is taken, although Prefetch Abort and Data Abort vector catch debug events are ignored. Debug exception entry is described in *Debug exception* on page 12-31. The Prefetch Abort handler can check the IFSR or the DSCR[5:2] bits, and the Data Abort handler can check the DFSR or the DSCR[5:2] bits, to find out the caused of the exception. If the cause was a Debug exception, the handler branches to the monitor target.

When the monitor target is running, it can determine and modify the processor state and new software debug events can be programmed.

12.11.2 Setting breakpoints, watchpoints, and vector catch debug events

When the monitor target is running, breakpoints, watchpoints, and vector catch debug events can be set. This can be done by executing MCR instructions to program the appropriate CP14 debug registers. The monitor target can only program these registers if the processor is in a privileged mode and Monitor debug-mode is selected and enabled, see *Debug Status and Control Register bit field definitions* on page 12-10.

You can program a vector catch debug event using CP14 Debug Vector Catch Register.

You can program a breakpoint debug event using CP14 Breakpoint Value Registers and CP14 Breakpoint Control Registers, see *CP14 c64-c69, Breakpoint Value Registers (BVR)* on page 12-15 and *CP14 c80-c85, Breakpoint Control Registers (BCR)* on page 12-17.

You can program a watchpoint debug event using CP14 Watchpoint Value Registers and CP14 Watchpoint Control Registers, see *CP14 c96-c97, Watchpoint Value Registers (WVR)* on page 12-20, and *CP14 c112-c113, Watchpoint Control Registers (WCR)* on page 12-20.

Setting a simple breakpoint on an IVA

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

1. Read the BCR.
2. Clear the BCR[0] enable breakpoint bit in the read word and write it back to the BCR. Now the breakpoint is disabled.
3. Write the IVA to the BVR register.
4. Write to the BCR with its fields set as follows:
   - BCR[21] meaning of BVR bit cleared, to indicate that the value loaded into BVR is to be compared against the IVA bus.
   - BCR[20] enable linking bit cleared, to indicate that this breakpoint is not to be linked.
   - BCR[8:5] byte address select BCR field as required.
   - BCR[2:1] supervisor access BCR field as required.
   - BCR[0] enable breakpoint bit set.

——— **Note** ———

Any BVR can be compared against the IVA bus.

———————————

Setting a simple breakpoint on a context ID value

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

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

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

Note

Any BVR can be compared against the IVA bus.

Setting a linked breakpoint

In the following sequence b is any of the breakpoint registers pairs with context ID comparison capability, and a is any of the implemented breakpoints different from b.

You can link IVA holding and contextID-holding breakpoints register pairs as follows:

1. Read the BCRa and BCRb.
2. Clear the BCRa[0] and BCRb[0] enable breakpoint bits in the read words and write them back to the BCRs. Now the breakpoints are disabled.
3. Write the IVA to the BVRa register.
4. Write the context ID to the BVRb register.
5. Write to the BCRb with its fields set as follows:
    - BCRb[21] meaning of BVR bit set, to indicate that the value loaded into BVRb is to be compared against the CP15 context ID register 13
    - BCRb[20] enable linking bit, set
    - BCRb[8:5] byte address select set to b1111
    - BCRb[2:1] supervisor access set to b11
    - BCRb[0] enable breakpoint bit set.
6. Write to the BCRa with its fields set as follows:
    - BCRa[21] meaning of BVR bit cleared, to indicate that the value loaded into BVRa is to be compared against the IVA bus

- BCRa[20] enable linking bit set, in order to link this BRP with the one indicated by BCRa[19:16] (BRPb in this example)
- binary representation of b into BCR[19:6] linked BRP field
- BCRa[8:5] byte address select field as required
- BCRa[2:1] supervisor access field as required
- BCRa[0] enable breakpoint set.

Setting a simple watchpoint

You can set a simple watchpoint as follows:

1. Read the WCR.
2. Clear the WCR[0] enable watchpoint bit in the read word and write it back to the WCR. Now the watchpoint is disabled.
3. Write the DVA to the WVR register.
4. Write to the WCR with its fields set as follows:
   - WCR[20] enable linking bit cleared, to indicate that this watchpoint is not to be linked
   - WCR byte address select, load/store access, and supervisor access fields as required
   - WCR[0] enable watchpoint bit set.

Note

Any WVR can be compared against the DVA bus.

Setting a linked watchpoint

In the following sequence b is any of the BRPs with context ID comparison capability. You can use any of the WRPs.

You can link WRPs and contextID-holding BRPs as follows:

1. Read the WCR and BCRb.
2. Clear the WCR[0] Enable watchpoint and the BCRb[0] Enable breakpoint bits in the read words and write them back to the WCR and BCRb. Now the watchpoint and the breakpoint are disabled.
3. Write the DVA to the WVR register.

4. Write the context ID to the BVRb register.
5. Write to the WCR with its fields set as follows:
   - WCR[20] enable linking bit set, in order to link this WRP with the BRP indicated by WCR[19:16] (BRPb in this example)
   - Binary representation of b into WCR[19:6] linked BRP field
   - WCR byte address select, load/store access, and supervisor access fields as required
   - WCR[0] enable watchpoint bit set.
6. Write to the BCRb with its fields set as follows:
   - BCRb[21] meaning of BVR bit set, to indicate that the value loaded into BVRb is to be compared against the CP15 Context ID Register.
   - BCRb[20] enable linking bit, set
   - BCRb[8:5] byte address select set to b1111
   - BCRb[2:1] supervisor access set to b11
   - BCRb[0] enable breakpoint bit set.

12.11.3 Setting software breakpoint debug events (BKPT)

To set a software breakpoint on a particular Virtual Address, the monitor target must perform the following steps:

1. Read memory location and save actual instruction.
2. Write BKPT instruction to the memory location.
3. Read memory location again to check that the BKPT instruction has been written.
4. If it has not been written, determine the reason.

Note

Cache coherency issues might arise when writing a BKPT instruction.

12.11.4 Using the debug communications channel

To read a word sent by a DBGTAP debugger:

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

3. Read the word from the rDTR, CP14 Data Transfer Register c5.

To write a word for a DBGTAP debugger:

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

12.12 Halt mode debugging

Halt mode is used to debug the ARM11 MPCore processor using external hardware connected to the DBGTAP. The external hardware provides an interface to a DBGTAP debugger application. You can only select Halt mode by setting the halt bit (bit 14) of the DSCR, which is only writable through the Debug Test Access Port. See Chapter 13 *Debug Test Access Port*.

In Halt mode the processor stops executing instructions if one of the following events occurs:

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

When the processor is halted, it is controlled by sending instructions to the integer unit through the DBGTAP. Any valid instruction can be scanned into the processor, and the effect of the instruction upon the integer unit is as if it was executed under normal operation. Also accessible through the DBGTAP is a register to transfer data between CP14 and the DBGTAP debugger.

The integer unit is restarted by executing a DBGTAP Restart instruction.

12.12.1 Entering debug state

When a debug event occurs and Halt mode is selected and enabled then the processor enters debug state as defined in *Debug state* on page 12-33.

When the core is in debug state, the DBGTAP debugger can determine and modify the processor state and new debug events can be programmed.

12.12.2 Exiting debug state

You can force the processor out of debug state using the DBGTAP Restart instruction. See *Exiting debug state* on page 13-5. The DSCR[1] core restarted bit indicates if the core has already returned to normal operation.

12.12.3 Programming debug events

In Halt mode debugging you can program the following debug events:

- *Setting breakpoints, watchpoints, and vector catch debug events* on page 12-46

- *Setting software breakpoints (BKPT)*
- *Reading and writing to memory*.

Setting breakpoints, watchpoints, and vector catch debug events

For setting breakpoints, watchpoints, and vector catch debug events when in Halt mode, the debug host has to use the same CP14 debug registers and the same sequence of operations as in Monitor debug-mode debugging (see *Setting breakpoints, watchpoints, and vector catch debug events* on page 12-39). The only difference is that the CP14 debug registers are accessed using the DBGTAP scan chains, see *DBGTAP controller overview* on page 13-6.

Note

A DBGTAP debugger can access the CP14 debug registers whether the processor is in debug state or not, so these debug events can be programmed while the processor is in ARM, Thumb, or Java state.

Setting software breakpoints (BKPT)

To set a software breakpoint, the DBGTAP debugger must perform the same steps as the monitor target (described in *Setting breakpoints, watchpoints, and vector catch debug events* on page 12-39). The difference is that CP14 debug registers are accessed using the DBGTAP scan chains, see Chapter 13 *Debug Test Access Port*.

Reading and writing to memory

See *Debug sequences* on page 13-33 for memory access sequences using the MPCore Debug Test Access Port.

12.13 External signals

The following external signals are used by debug:

DBGACK	Debug acknowledge signal. The processor asserts this output signal to indicate the system has entered Debug state. See *Debug state* on page 12-33 for a definition of the Debug state.
DBGEN	Debug enable signal. When this signal is LOW, DSCR[15:14] is read as 0 and the processor behaves as if in debug disabled mode.
EDBGRQ	External debug request signal. As described in *External debug request signal* on page 12-28, this input signal forces the core into Debug state if the debug logic is in Halt mode.
DBGNOPWRDWN	Powerdown disable signal generated from DSCR[9]. When this signal is HIGH, the system power controller is forced into Emulate mode. This is to avoid losing CP14 debug state that can only be written through the DBGTAP. Therefore, DSCR[9] must only be set if Halt mode debugging is necessary.

Chapter 13
Debug Test Access Port

This chapter introduces the Debug Test Access Port built into the MP11 CPUs within the ARM11 MPCore processor. It contains the following sections:

13.1 Debug Test Access Port and Halt mode

JTAG-based hardware debug using Halt mode provides access to the ARM11 MPCore processor CPUs and their debug units. Access is through scan chains and the *Debug Test Access Port* (DBGTAP). Figure 13-1 shows the *DBGTAP State Machine* (DBGTAPSM).

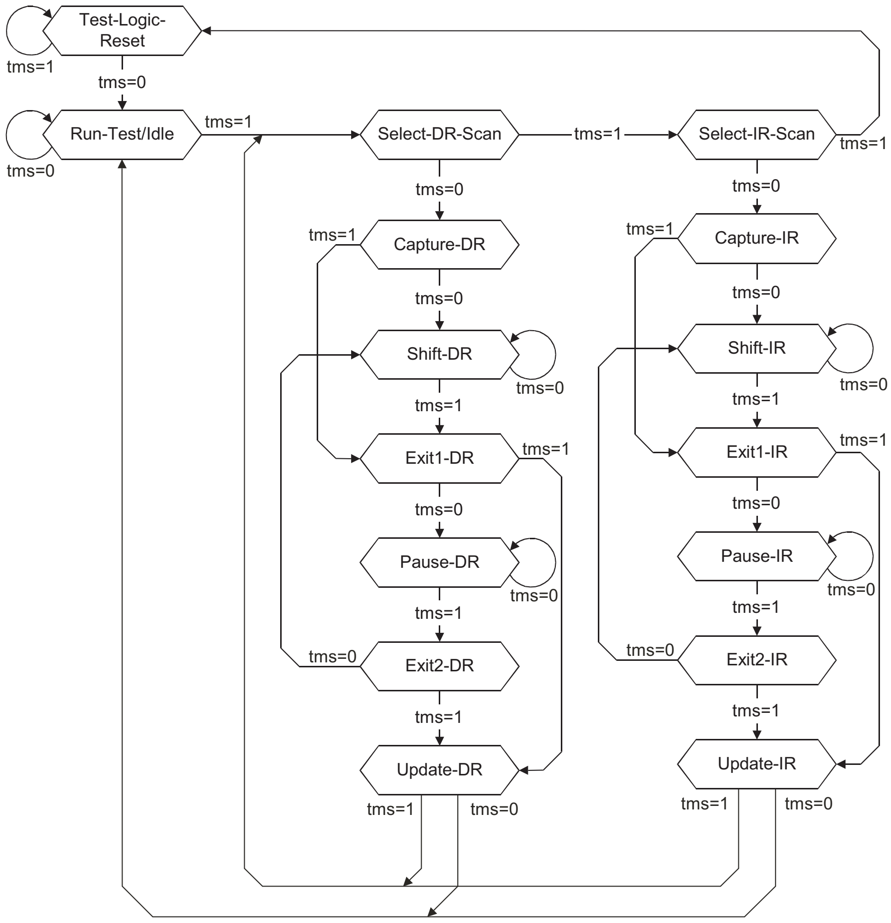


Figure 13-1 JTAG DBGTAP state machine diagram

From IEEE Std 1149.1-1990. Copyright 2002 - 2004 IEEE. All rights reserved.

13.2 Synchronizing RealView ICE

The system and test clocks are synchronized within the ARM11 MPCore processor and so no additional logic is required to synchronize off-chip debug clocking. The ARM11 MPCore processor provides a synchronized version of the **TDI** debug signal through **DBGTDI** output. This enables the implementor to daisy-chain MP11 CPU debug scan chains as required. See *Daisy-chaining debug signals* on page 11-14 for an example.

The interface to RealView ICE consists of the following signals:

- The **TDO** debug signal of the last MP11 CPU in the debug scan chain
- **TCK**
- **nTRST**
- **TDI**
- **TMS**
- **RTCK**.

13.3 Entering debug state

Halt mode is enabled by writing a 1 to bit 14 of the DSCR, see *CP14 c1, Debug Status and Control Register (DSCR)* on page 12-9. This can only be done by a DBGTAP debugger hardware such as RealView ICE. When this mode is enabled the processor halts, instead of taking an exception in software, if one of the following events occurs:

- A Halt instruction is scanned in through the DBGTAP. The DBGTAP controller must pass through Run-Test/Idle to issue the Halt command to the ARM.
- A vector catch occurs.
- A breakpoint hits.
- A watchpoint hits.
- A BKPT instruction is executed.
- **EDBGRQ** is asserted.

The core halted bit in the DSCR is set when debug state is entered. At this point, the debugger determines why the integer unit was halted and preserves the processor state. The MSR instruction can be used to change modes and gain access to all banked registers in the machine. While in debug state:

- the PC is not incremented
- interrupts are ignored
- all instructions are read from the instruction transfer register (scan chain 4).

Debug state is described in *Debug state* on page 12-33.

13.4 Exiting debug state

To exit from debug state, scan in the Restart instruction through the MPCore DBGTAP. You might want to adjust the PC before restarting, depending on the way the integer unit entered debug state. When the state machine enters the Run-Test/Idle state, normal operations resume. The delay, waiting until the state machine is in Run-Test/Idle, enables conditions to be set up in other devices in a multiprocessor system without taking immediate effect. When Run-Test/Idle state is entered, all the processors resume operation simultaneously. The core restarted bit is set when the Restart sequence is complete.

13.5 DBGTAP controller overview

The MP11 CPU DBGTAP controller is the part of the debug unit that enables access through the DBGTAP to the on-chip debug resources, such as breakpoint and watchpoint registers. The DBGTAP controller is based on the IEEE 1149.1 standard and supports:

- a device ID register
- a Bypass Register
- a five-bit Instruction Register
- a five-bit Scan Chain Select Register.

In addition, the public instructions listed in Table 13-1 are supported.

Table 13-1 Supported public instructions

Binary code	Instruction	Description
b00000	EXTEST	This instruction connects the selected scan chain between **CPUTDI** and **TDO**. When the Instruction Register is loaded with the EXTEST instruction, the debug scan chains can be written. See *Scan chains* on page 13-11.
b00001	-	Reserved.
b00010	Scan_N	Selects the Scan Chain Select Register (SCREG). This instruction connects SCREG between **CPUTDI** and **TDO**. See *Scan chain select register (SCREG)* on page 13-10.
b00011	-	Reserved.
b00100	Restart	Forces the processor to leave debug state. This instruction is used to exit from debug state. The processor restarts when the Run-Test/Idle state is entered.
b00101	-	Reserved.
b00110	-	Reserved.
b00111	-	Reserved.
b01000	Halt	Forces the processor to enter debug state. This instruction is used to stop the integer unit and put it into debug state. The core can only be put into debug state if Halt mode is enabled.
b01001	-	Reserved.
b01010-b01011	-	Reserved.
b01100	INTEST	This instruction connects the selected scan chain between **CPUTDI** and **TDO**. When the instruction register is loaded with the INTEST instruction, the debug scan chains can be read. See *Scan chains* on page 13-11.

Table 13-1 Supported public instructions (continued)

Binary code	Instruction	Description
b01101-b11100	-	Reserved.
b11101	ITRsel	When this instruction is loaded into the IR (Update-DR state), the DBGTAP controller behaves as if IR=EXTEST and SCREG=4. The ITRsel instruction makes the DBGTAP controller behave as if EXTEST and scan chain 4 are selected. It can be used to speed up certain debug sequences. See *Using the ITRsel IR instruction* on page 13-24 for the effects of using this instruction.
b11110	IDcode	See IEEE 1149.1. Selects the DBGTAP controller Device ID Code Register. The IDcode instruction connects the Device ID Code Register (or ID register) between **CPUTDI** and **TDO**. The ID register is a 32-bit register that enables you to determine the manufacturer, part number, and version of a component using the DBGTAP. See *Device ID code register* on page 13-9 for details of selecting and interpreting the ID register value.
b11111	Bypass	See IEEE 1149.1. Selects the DBGTAP controller Bypass Register. The Bypass instruction connects a 1-bit shift register (the Bypass Register) between **CPUTDI** and **TDO**. The first bit shifted out is a 0. All unused DBGTAP controller instruction codes default to the Bypass instruction. See *Bypass register* on page 13-8.

Note

Sample/Preload, Clamp, HighZ, and ClampZ instructions are not implemented because the MPCore DBGTAP controller does not support the attachment of external boundary scan chains.

All unused DBGTAP controller instructions default to the Bypass instruction.

13.6 Debug registers

You can connect the following debug registers or scan chains between **DBGTDI** and **DBGTDO**:

- *Bypass register*
- *Device ID code register* on page 13-9
- *Instruction Register* on page 13-9
- *Scan chain select register (SCREG)* on page 13-10
- *Scan chain 0, debug ID register (DIDR)* on page 13-11
- *Scan chain 1, Debug Status and Control Register (DSCR)* on page 13-12
- *Scan chain 4, Instruction Transfer Register (ITR)* on page 13-13
- *Scan chain 5* on page 13-15
- *Scan chain 7* on page 13-18.

13.6.1 Bypass register

Purpose	Bypasses the device by providing a path between **CPUTDI** and **TDO**.
Length	1 bit.
Operating mode	When the bypass instruction is the current instruction in the Instruction Register, serial data is transferred from **CPUTDI** to **TDO** in the Shift-DR state with a delay of one **TCK** cycle. There is no parallel output from the Bypass Register. A logic 0 is loaded from the parallel input of the Bypass Register in the Capture-DR state. Nothing happens at the Update-DR state.
Order	Figure 13-2 shows the order of bits in the Bypass Register.

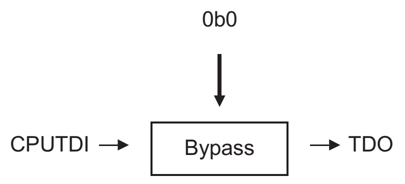


Figure 13-2 Bypass register bit order

13.6.2 Device ID code register

Purpose Device identification. To distinguish the MP11 CPUs from other ARM processors, the DBGTAP controller ID is unique for each. This means that a DBGTAP debugger such as RealView ICE can easily see which processor it is connected to. The Device ID Register version and manufacturer ID fields are routed to the edge of the chip so that partners can create their own Device ID numbers by tying the pins to HIGH or LOW values.

The default manufacturer ID for the MP11 CPUs is 0x23B.

The part number field is hard-wired inside the MP11 CPUs to `0x7B37`. See *c0, ID Code Register* on page 3-12 for details on how ARM semiconductor partner-specific are identified.

Length 32 bits

Operating mode When the ID code instruction is current, the shift section of the Device ID Code Register is selected as the serial path between **CPUTDI** and **TDO**. There is no parallel output from the ID register. The 32-bit device ID code is loaded into this shift section during the Capture-DR state. This is shifted out during Shift-DR (least significant bit first) while a *don't care* value is shifted in. The shifted-in data is ignored in the Update-DR state.

Order Figure 13-3 shows the bit order in the ID code register.

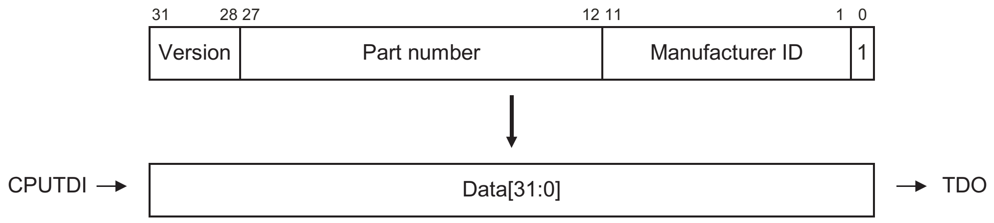


Figure 13-3 Device ID code register bit order

13.6.3 Instruction Register

Purpose Holds the current DBGTAP controller instruction.

Length 5 bits.

Operating mode When in Shift-IR state, the shift section of the Instruction Register is selected as the serial path between **CPUTDI** and **TDO**. At the Capture-IR state, the binary value b00001 is loaded into this shift section. This is shifted out during Shift-IR (least significant bit first), while a new instruction is shifted in (least significant bit first). At the Update-IR state, the value in the shift section is loaded into the Instruction Register so it becomes the current instruction. On DBGTAP reset, the IDcode becomes the current instruction.

Order Figure 13-4 shows the bit order in the Instruction Register.

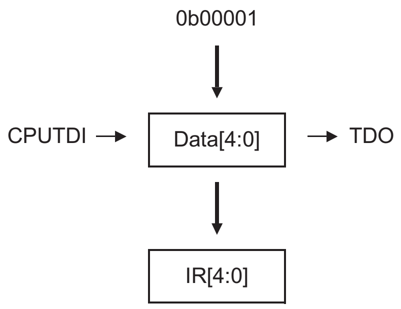


Figure 13-4 Instruction Register bit order

13.6.4 Scan chain select register (SCREG)

Purpose Holds the currently active scan chain number.

Length 5 bits.

Operating mode After Scan_N has been selected as the current instruction, when in Shift-DR state, the shift section of the Scan Chain Select Register is selected as the serial path between **CPUTDI** and **TDO**. At the Capture-DR state, the binary value b10000 is loaded into this shift section. This is shifted out during Shift-DR (least significant bit first), while a new value is shifted in (least significant bit first). At the Update-DR state, the value in the shift section is loaded into the Scan Chain Select Register to become the current active scan chain. All further instructions such as INTEST then apply to that scan chain. The currently selected scan chain only changes when a Scan_N or ITRsel instruction is executed, or a DBGTAP reset occurs. On DBGTAP reset, scan chain 3 is selected as the active scan chain.

Order Figure 13-5 on page 13-11 shows the bit order in the Scan Chain Select Register.

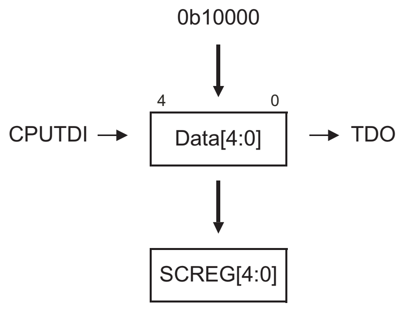


Figure 13-5 Scan Chain Select Register bit order

13.6.5 Scan chains

To access the debug scan chains you must:

1. Load the Scan_N instruction into the IR. Now SCREG is selected between **CPUTDI** and **TDO**.
2. Load the number of the desired scan chain. For example, load b00101 to access scan chain 5.
3. Load either INTEST or EXTEST into the IR.
4. Go through the DR leg of the DBGTAPSM to access the scan chain.

You must use INTEST and EXTEST as follows:

INTEST Use INTEST for reading the active scan chain. Data is captured into the shift register at the Capture-DR state. The previous value of the scan chain is shifted out during the Shift-DR state, while a new value is shifted in. The scan chain is not updated during Update-DR. Those bits or fields that are defined as cleared on read are only cleared if INTEST is selected, even when EXTEST also captures their values.

EXTEST Use EXTEST for writing the active scan chain. Data is captured into the shift register at the Capture-DR state. The previous value of the scan chain is shifted out during the Shift-DR state, while a new value is shifted in. The scan chain is updated with the new value during Update-DR.

Scan chain 0, debug ID register (DIDR)

Purpose Debug.

Length 8 + 32 = 40 bits.

Description Debug identification. This scan chain accesses CP14 debug register 0, the debug ID register. Additionally, the eight most significant bits of this scan chain contain an implementor code. This field is hardwired to `0x41`, the implementor code for ARM Limited, as specified in the *ARM Architecture Reference Manual*. This register is read-only. Therefore, EXTEST has the same effect as INTEST.

Order Figure 13-6 shows the bit order in scan chain 0.

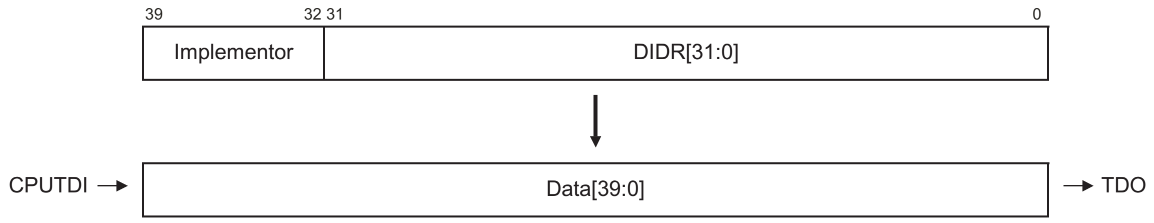


Figure 13-6 Scan chain 0 bit order

Scan chain 1, Debug Status and Control Register (DSCR)

Purpose Debug.

Length 32 bits.

Description This scan chain accesses CP14 register 1, the DSCR. This is mostly a read/write register, although certain bits are read-only for the Debug Test Access Port. See *CP14 c1, Debug Status and Control Register (DSCR)* on page 12-9 for details of DSCR bit definitions, and for read/write attributes for each bit. Those bits defined as cleared on read are only cleared if INTEST is selected.

Order Figure 13-7 on page 13-13 shows the bit order in scan chain 1.

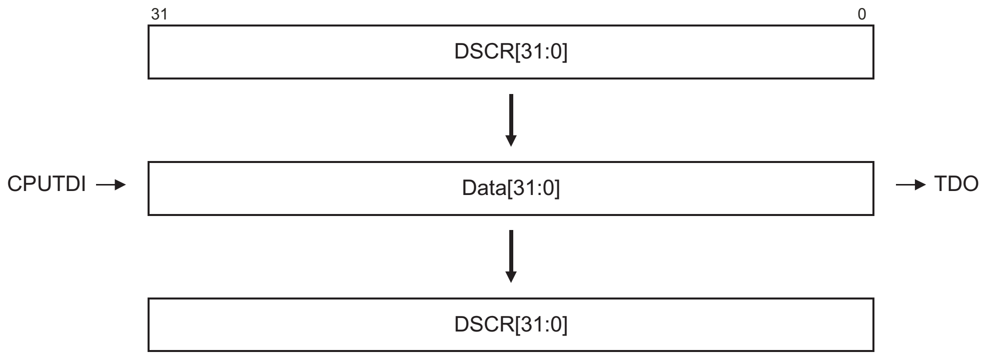


Figure 13-7 Scan chain 1 bit order

The following DSCR bits affect the operation of other scan chains:

DSCR[30:29]	rDTRfull and wDTRfull flags. These indicate the status of the rDTR and wDTR registers. They are copies of the rDTRempty (NOT rDTRfull) and wDTRfull bits that the DBGTAP debugger sees in scan chain 5.
DSCR[13]	Execute ARM instruction enable bit. This bit enables the mechanism used for executing instructions in debug state. It changes the behavior of the rDTR and wDTR registers, the sticky precise Data Abort bit, rDTRempty, wDTRfull, and InstCompl flags. See *Scan chain 5* on page 13-15.
DSCR[6]	Sticky precise Data Abort flag. If the core is in debug state and the DSCR[13] execute ARM instruction enable bit is HIGH, then this flag is set on precise Data Aborts. See *CP14 c1, Debug Status and Control Register (DSCR)* on page 12-9.

——— **Note** ———

Unlike DSCR[6], DSCR [7] sticky imprecise Data Aborts flag does not affect the operation of the other scan chains.

———————————

Scan chain 4, Instruction Transfer Register (ITR)

Purpose	Debug
Length	1 + 32 = 33 bits

Description This scan chain accesses the *Instruction Transfer Register* (ITR), used to send instructions to the core through the *Prefetch Unit* (PU). It consists of 32 bits of information, plus an additional bit to indicate the completion of the instruction sent to the core (InstCompl). The InstCompl bit is read-only.

While in debug state, an instruction loaded into the ITR can be issued to the core by making the DBGTAPSM go through the Run-Test/Idle state. The InstCompl flag is cleared when the instruction is issued to the core and set when the instruction completes.

For an instruction to be issued when going through Run-Test/Idle state, you must ensure the following conditions are met:

- The processor must be in debug state.
- The DSCR[13] execute ARM instruction enable bit must be set. For details of the DSCR see *CP14 c1, Debug Status and Control Register (DSCR)* on page 12-9.
- Scan chain 4 or 5 must be selected.
- INTEST or EXTEST must be selected.
- Ready flag must be captured set. That is, the last time the DBGTAPSM went through Capture-DR the InstCompl flag must have been set.
- The DSCR[6] sticky precise Data Abort flag must be clear. This flag is set on precise Data Aborts.

For an instruction to be loaded into the ITR when going through Update-DR, you must ensure the following conditions are met:

- The processor can be in any state.
- The value of DSCR[13] execute ARM instruction enable bit does not matter.
- Scan chain 4 must be selected.
- EXTEST must be selected.
- Ready flag must be captured set. That is, the last time the DBGTAPSM went through Capture-DR the InstCompl flag must have been set.
- The value of DSCR[6] sticky precise Data Abort flag does not matter.

Order Figure 13-8 on page 13-15 shows the bit order in scan chain 4.

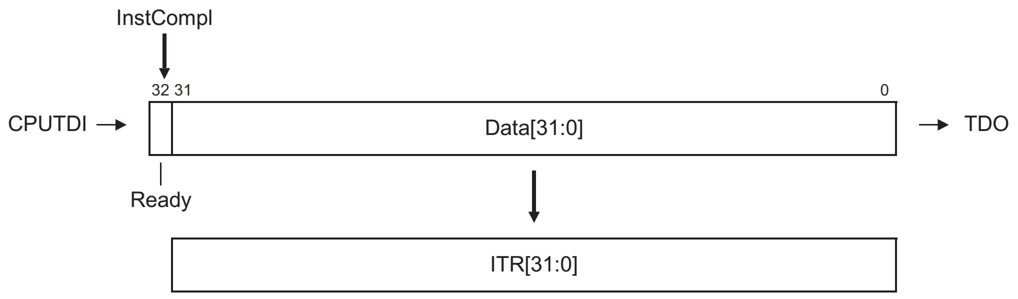


Figure 13-8 Scan chain 4 bit order

It is important to distinguish between the InstCompl flag and the Ready flag:

- The InstCompl flag signals the completion of an instruction.
- The Ready flag is the captured version of the InstCompl flag, captured at the Capture-DR state. The Ready flag conditions the execution of instructions and the update of the ITR.

The following points apply to the use of scan chain 4:

- When an instruction is issued to the core in debug state, the PC is not incremented. It is only changed if the instruction being executed explicitly writes to the PC. For example, branch instructions and move to PC instructions.
- If CP14 debug register c5 is a source register for the instruction to be executed, the DBGTAP debugger must set up the data in the rDTR before issuing the coprocessor instruction to the core. See *Scan chain 5*.
- Setting DSCR[13] the execute ARM instruction enable bit when the core is not in debug state leads to Unpredictable behavior.
- The ITR is write-only. When going through the Capture-DR state, an Unpredictable value is loaded into the shift register.

Scan chain 5

Purpose	Debug.
Length	1 + 1 + 32 = 34 bits.
Description	This scan chain accesses CP14 register c5, the data transfer registers, rDTR and wDTR. The rDTR is used to transfer words from the DBGTAP debugger to the core, and is read-only to the core and write-only to the

DBGTAP debugger. The wDTR is used to transfer words from the core to the DBGTAP debugger, and is read-only to the DBGTAP debugger and write-only to the core.

The DBGTAP controller only sees one (read/write) register through scan chain 5, and the appropriate register is chosen depending on the instruction used. INTEST selects the wDTR, and EXTEST selects the rDTR.

Additionally, scan chain 5 contains some status flags. These are nRetry, Valid, and Ready, which are the captured versions of the rDTRempty, wDTRfull, and InstCompl flags respectively. All are captured at the Capture-DR state.

Order Figure 13-9 shows the bit order in scan chain 5 with EXTEST selected Figure 13-10 shows the bit order in scan chain 5 with INTEST selected.

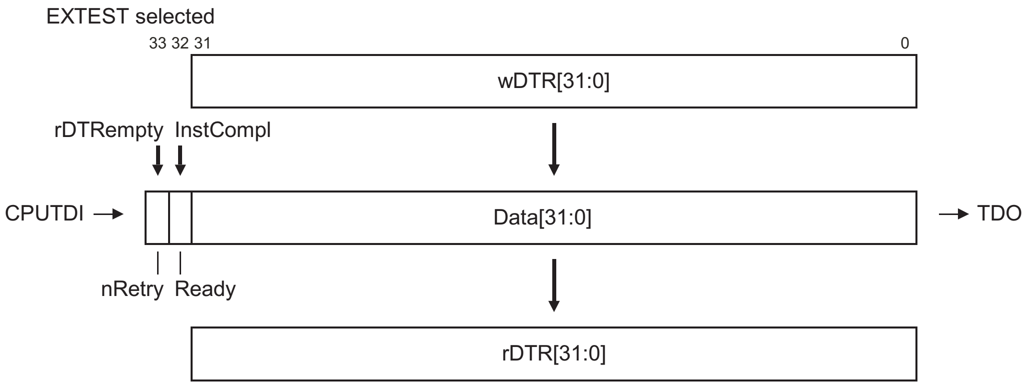


Figure 13-9 Scan chain 5 bit order, EXTEST selected

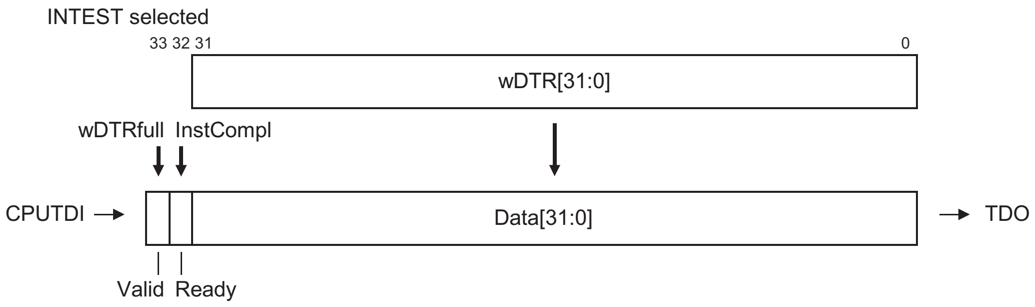


Figure 13-10 Scan chain 5 bit order, INTEST selected

You can use scan chain 5 for two purposes:

- As part of the *Debug Communications Channel* (DCC). The DBGTAP debugger uses scan chain 5 to exchange data with software running on the core. The software accesses the rDTR and wDTR using coprocessor instructions.
- For examining and modifying the processor state while the core is halted. For example, to read the value of an ARM register:
    1. Issue an `MCR cp14, 0, Rd, c0, c5, 0` instruction to the core to transfer the register contents to the CP14 debug c5 register.
    2. Scan out the wDTR.

The DBGTAP debugger can use the DSCR[13] execute ARM instruction enable bit to indicate to the core that it is going to use scan chain 5 as part of the DCC or for examining and modifying the processor state. DSCR[13] = 0 indicates DCC use. The behavior of the rDTR and wDTR registers, the sticky precise Data Abort, rDTRempty, wDTRfull, and InstCompl flags changes accordingly:

- DSCR[13] = 0:
    - The wDTRfull flag is set when the core writes a word of data to the DTR and cleared when the DBGTAP debugger goes through the Capture-DR state with INTEST selected. Valid indicates the state of the wDTR register, and is the captured version of wDTRfull. Although the value of wDTR is captured into the shift register, regardless of INTEST or EXTEST, wDTRfull is only cleared if INTEST is selected.
    - The rDTR empty flag is cleared when the DBGTAP debugger writes a word of data to the rDTR, and set when the core reads it. nRetry is the captured version of rDTRempty.
    - rDTR overwrite protection is controlled by the nRetry flag. If the nRetry flag is sampled clear, meaning that the rDTR is full, when going through the Capture-DR state, then the rDTR is not updated at the Update-DR state.
    - The InstCompl flag is always set.
    - The sticky precise Data Abort flag is Unpredictable. See *CP14 c1, Debug Status and Control Register (DSCR)* on page 12-9.
- DSCR[13] = 1:
    - The wDTRfull flag behaves as if DSCR[13] is clear. However, the Ready flag can be used for handshaking in this mode.
    - The rDTR empty flag status behaves as if DSCR[13] is clear. However, the Ready flag can be used for handshaking in this mode.

    — rDTR overwrite protection is controlled by the Ready flag. If the InstCompl flag is sampled clear when going through Capture-DR, then the rDTR is not updated at the Update-DR state. This prevents an instruction that uses the rDTR as a source operand from having it modified before it has time to complete.
    — The InstCompl flag changes from 1 to 0 when an instruction is issued to the core, and from 0 to 1 when the instruction completes execution.
    — The sticky precise Data Abort flag is set on precise Data Aborts.

The behavior of the rDTR and wDTR registers, the sticky precise Data Abort, rDTRempty, wDTRfull, and InstCompl flags when the core changes state is as follows:

- The DSCR[13] execute ARM instruction enable bit must be clear when the core is not in debug state. Otherwise, the behavior of the rDTR and wDTR registers, and the flags, is Unpredictable.
- When the core enters debug state, none of the registers and flags are altered.
- When the DSCR[13] execute ARM instruction enable bit is changed from 0 to 1:
    1. None of the registers and flags are altered.
    2. Ready flag can be used for handshaking.
- The InstCompl flag must be set when the DSCR[13] execute ARM instruction enable bit is changed from 1 to 0. Otherwise, the behavior of the core is Unpredictable. If the DSCR[13] flag is cleared correctly, none of the registers and flags are altered.
- When the core leaves debug state, none of the registers and flags are altered.

Scan chain 7

Purpose	Debug.
Length	7 + 32 + 1 = 40 bits.
Description	Scan chain 7 accesses the VCR, PC, BRPs, and WRPs. The accesses are performed with the help of read or write request commands. A read request copies the data held by the addressed register into scan chain 7. A write request copies the data held by the scan chain into the addressed register. When a request is finished the ReqCompl flag is set. The DBGTAP debugger must poll it and check it is set before another request can be issued.

The exact behavior of the scan chain is as follows:

- Either EXTEST or INTEST have to be selected. They have the same meaning in this scan chain.
- If the value captured by the Ready/nRW bit at the Capture-DR state is 1, the data that is being shifted in generates a request at the Update-DR state. The Address field indicates the register being accessed (see Table 13-2 on page 13-20), the Data field contains the data to be written and the Ready/nRW bit holds the read/write information (0=read and 1=write). If the request is a read, the Data field is ignored.
- When a request is placed, the Address and Data sections of the scan chain are frozen. That is, their contents are not shifted until the request is completed. This means that, if the value captured in the Ready/nRW field at the Capture-DR state is 0, the shifted-in data is ignored and the shifted-out value is all 0s.
- After a read request has been placed, if the DBGTAPSM goes through the Capture-DR state and a logic 1 is captured in the Ready/nRW field, this means that the shift register has also captured the requested register contents. Therefore, they are shifted out at the same time as the Ready/nRW bit. The Data field is corrupted as new data is shifted in.
- After a write request has been placed, if the DBGTAPSM goes through the Capture-DR state and a logic 1 is captured in the Ready/nRW field, this means that the requested write has completed successfully.
- If the Address field is all 0s (address of the NULL register) at the Update-DR state, then no request is generated.
- A request to a reserved register generates Unpredictable behavior.

Order Figure 13-11 shows the bit order in scan chain 7.

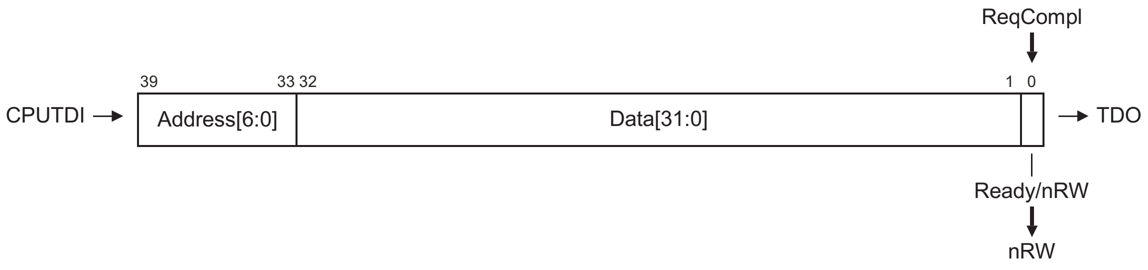


Figure 13-11 Scan chain 7 bit order

A typical sequence for writing registers is as follows:

1. Scan in the address of a first register, the data to write, and a 1 to indicate that this is a write request.

2. Scan in the address of a second register, the data to write, and a 1 to indicate that this is a write request.

   Scan out 40 bits. If Ready/nRW is 0 repeat this step. If Ready/nRW is 1, the first write request has completed successfully and the second has been placed.

3. Scan in the address 0. The rest of the fields are not important.

   Scan out 40 bits. If Ready/nRW is 0 repeat this step. If Ready/nRW is 1, the second write request has completed successfully. The scanned-in null request has avoided the generation of another request.

A typical sequence for reading registers is as follows:

1. Scan in the address of a first register and a 0 to indicate that this is a read request. The Data field is not important.

2. Scan in the address of a second register and a 0 to indicate that this is a read request.

   Scan out 40 bits. If Ready/nRW is 0 then repeat this step. If Ready/nRW is 1, the first read request has completed successfully and the next scanned-out 32 bits are the requested value. The second read request was placed at the Update-DR state.

3. Scan in the address 0 (the rest of the fields are not important).

   Scan out 40 bits. If Ready/nRW is 0 then repeat this step. If Ready/nRW is 1, the second read request has completed successfully and the next scanned-out 32 bits are the requested value. The scanned-in null request has avoided the generation of another request.

Table 13-2 shows the register map scan chain 7 register map. This is similar to the CP14 debug register map.

Table 13-2 Scan chain 7 register map

Address[6:0]	Register number	Abbreviation	Register name
b0000000	0	NULL	No request register
b0000001-b0000110	1-6	-	Reserved
b0000111	7	VCR	Vector Catch Register

Table 13-2 Scan chain 7 register map (continued)

Address[6:0]	Register number	Abbreviation	Register name
b0001000	8	PC	Program counter
b0010011-b0111111	19-63	-	Reserved
b1000000-b1000101	64-69	BVRy[a]	Breakpoint Value Registers
b1000110-b1001111	70-79	-	Reserved
b1010000-b1010101	80-85	BCRy[a]	Breakpoint Control Registers
b1010110-b1011111	86-95	-	Reserved
b1100000-b1100001	96-97	WVRy[a]	Watchpoint Value Registers
b1100010-1b101111	98-111	-	Reserved
b1110000-b1110001	112-113	WCRy[a]	Watchpoint Control Registers
b1110010-b1111111	114-127	-	Reserved

a. y is the decimal representation for the binary number Address[3:0]

The following points apply to the use of scan chain 7:

- Every time there is a request to read the PC, a sample of its value is copied into scan chain 7. Writes are ignored. The sampled value can be used for profiling of the code. See *Interpreting the PC samples* on page 13-22 for details of how to interpret the sampled value.
- When accessing registers using scan chain 7, the processor can be either in debug state or in normal state. This implies that breakpoints, watchpoints, and vector catches can be programmed through the Debug Test Access Port even if the processor is running. However, although a PC read can be requested in debug state, the result is Undefined.

Interpreting the PC samples

The PC values read correspond to instructions committed for execution, including those that failed their condition code. However, these values are offset as described in Table 12-15 on page 12-29. These offsets are different for different processor states, so additional information is required:

- If a read request to the PC completes and Data[1:0] equals b00, the read value corresponds to an ARM state instruction whose 30 most significant bits of the offset address (instruction address + 8) are given in Data[31:2].
- If a read request to the PC completes and Data[0] equals b1, the read value corresponds to a Thumb state instruction whose 31 most significant bits of the offset address (instruction address + 4) are given in Data[31:1].
- If a read request to the PC completes and Data[1:0] equals b10, the read value corresponds to a Java state instruction whose 30 most significant bits of its address are given in Data[31:2] (the offset is 0). Because of the state encoding, the lower two bits of the Java address are not sampled. However, the information provided is enough for profiling the code.
- If the PC is read while the processor is in Debug state, the result is Unpredictable.

Scan chains 8-15

These scan chains are reserved.

Scan chains 16-31

These scan chains are unassigned.

13.6.6 Reset

The DBGTAP is reset either by asserting **DBGnTRST**, or by clocking it while the DBGTAPSM is in the Test-Logic-Reset state. The processor, including CP14 debug logic, is not affected by these events. See *Reset modes* on page 11-4 and *CP14 registers reset* on page 12-23 for details.

13.7 Using the Debug Test Access Port

This section contains the following subsections:

13.7.1 Entering and leaving debug state

These debug sequences are described in detail in *Debug sequences* on page 13-33.

13.7.2 Executing instructions in debug state

When the ARM11 MPCore processor is in debug state, it can be forced to execute ARM state instructions using the DBGTAP. Two registers are used for this purpose, the *Instruction Transfer Register* (ITR) and the *Data Transfer Register* (DTR).

The ITR is used to insert an instruction into the processor pipeline. An ARM state instruction can be loaded into this register using scan chain number 4. When the instruction is loaded, and INTEST or EXTEST is selected, and scan chain 4 or 5 is selected, the instruction can be issued to the core by making the DBGTAPSM go through the Run-Test/Idle state, provided certain conditions are met (described in this section). This mechanism enables re-executing the same instruction over and over without having to reload it.

The DTR can be used in conjunction with the ITR to transfer data in and out of the core. For example, to read out the value of an ARM register:

1. issue an `MCR p14,0,<Rd>,c0,c5,0` instruction to the core to transfer the Rd contents to the c5 register
2. scan out the wDTR.

The DSCR[13] execute ARM instruction enable bit controls the activation of the ARM instruction execution mechanism. If this bit is cleared, no instruction is issued to the core when the DBGTAPSM goes through Run-Test/Idle. Setting this bit while the core is not in debug state leads to Unpredictable behavior. If the core is in debug state and this bit is set, the Ready and the sticky precise Data Abort flags condition the updates of the ITR and the instruction issuing as described in *Scan chain 4, Instruction Transfer Register (ITR)* on page 13-13.

As an example, this sequence stores out the contents of the ARM register r0:

1. Scan_N into the IR.
2. 1 into the SCREG.
3. INTEST into the IR.
4. Scan out the contents of the DSCR. This action clears the sticky precise Data Abort and sticky imprecise Data Abort flags.
5. EXTEST into the IR.
6. Scan in the previously read value with the DSCR[13] execute ARM instruction enable bit set.
7. Scan_N into the IR.
8. 4 into the SCREG.
9. EXTEST into the IR.
10. Scan the `MCR p14,0,R0,c0,c5,0` instruction into the ITR.
11. Go through the Run-Test/Idle state of the DBGTAPSM.
12. Scan_N into the IR.
13. 5 into the SCREG.
14. INTEST into the IR.
15. Scan out 34 bits. The 33rd bit indicates if the instruction has completed. If the bit is clear, repeat this step again.
16. The least significant 32 bits hold the contents of r0.

13.7.3 Using the ITRsel IR instruction

When the ITRsel instruction is loaded into the IR, at the Update-IR state, the DBGTAP controller behaves as if EXTEST and scan chain 4 are selected, but SCREG retains its value. It can be used to speed up certain debug sequences.

Figure 13-12 on page 13-25 shows the effect of the ITRsel IR instruction.

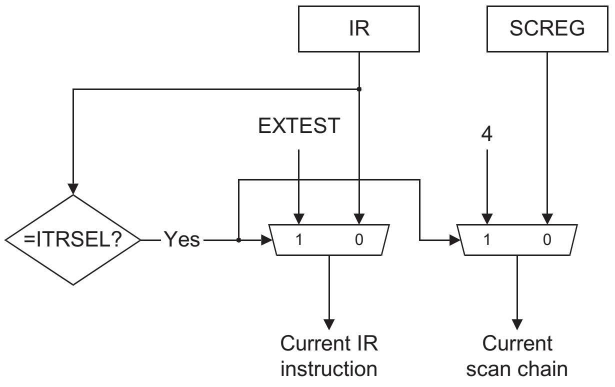


Figure 13-12 Behavior of the ITRsel IR instruction

Consider for example the preceding sequence to store out the contents of ARM register r0. This is the same sequence using the ITRsel instruction:

1. Scan_N into the IR.
2. 1 into the SCREG.
3. INTEST into the IR.
4. Scan out the contents of the DSCR. This action clears the sticky precise Data Abort and sticky imprecise Data Abort flags.
5. EXTEST into the IR.
6. Scan in the previously read value with the DSCR[13] execute ARM instruction enable bit set.
7. Scan_N into the IR.
8. 5 into the SCREG.
9. ITRsel into the IR. Now the DBGTAP controller works as if EXTEST and scan chain 4 is selected.
10. Scan the `MCR p14,0,R0,c0,c5,0` instruction into the ITR.
11. Go through the Run-Test/Idle state of the DBGTAPSM.
12. INTEST into the IR. Now INTEST and scan chain 5 are selected.
13. Scan out 34 bits. The 33rd bit indicates if the instruction has completed. If the bit is clear, repeat this step again.

14. The least significant 32 bits hold the contents of r0.

The number of steps has been reduced from 16 to 14. However, the bigger reduction comes when reading additional registers. Using the ITRsel instruction there are 6 extra steps (9 to 14), compared with 10 extra steps (7 to 16) in the first sequence.

13.7.4 Transferring data between the host and the core

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

- using the DCC, when the ARM11 MPCore processor is not in debug state
- using the instruction execution mechanism described in *Executing instructions in debug state* on page 13-23, when the core is in debug state.

This is described in:

- *Using the debug communications channel.*
- *Target to host debug communications channel sequence* on page 13-27
- *Host to target debug communications channel* on page 13-28
- *Transferring data in debug state* on page 13-28
- *Example sequences* on page 13-29.

13.7.5 Using the debug communications channel

The DCC is defined as the set of resources that the external DBGTAP debugger uses to communicate with a piece of software running on the core.

The DCC in the ARM11 MPCore processor is implemented using the two physically separate DTRs and a full/empty bit pair to augment each register, creating a bidirectional data port. One register can be read from the DBGTAP and is written from the processor. The other register is written from the DBGTAP and read by the processor. The full/empty bit pair for each register is automatically updated by the debug unit hardware, and is accessible to both the DBGTAP and to software running on the processor.

At the core side, the DCC resources are the following:

- CP14 Debug Transfer Register c5 (DTR). Data coming from a DBGTAP debugger can be read by an MRC or STC instruction addressed to this register. The core can write to this register any data intended for the DBGTAP debugger, using an MCR or LDC instruction. Because the DTR comprises both a read (rDTR) and a write portion (wDTR), a piece of data written by the core and another coming from the DBGTAP debugger can be held in this register at the same time.

- Some flags and control bits in CP14 Debug Status and Control Register c1 (DSCR):

DSCR[12]	User mode access to DCC disable bit. If this bit is set, only privileged software can access the DCC. That is, access the DSCR and the DTR.
DSCR[29]	The wDTRfull flag. When clear, this flag indicates to the core that the wDTR is ready to receive data from the core.
DSCR[30]	The rDTRfull flag. When set, this flag indicates to the core that there is data available to read at the DTR.

At the DBGTAP side, the resources are the following:

- Scan chain 5 (see *Scan chain 5* on page 13-15). The only part of this scan chain that it is not used for the DCC is the Ready flag. The rest of the scan chain is to be used in the following way:

rDTR	When the DBGTAPSM goes through the Update-DR state with EXTEST and scan chain 5 selected, and the nRetry flag set, the contents of the Data field are loaded into the rDTR. This is how the DBGTAP debugger sends data to the software running on the core.
wDTR	When the DBGTAPSM goes through the Capture-DR state with INTEST and scan chain 5 selected, the contents of the wDTR are loaded into the Data field of the scan chain. This is how the DBGTAP debugger reads the data sent by the software running on the core.
Valid flag	When set, this flag indicates to the DBGTAP debugger that the contents of the wDTR that it has just captured are valid.
nRetry flag	When set, this flag indicates to the DBGTAP debugger that the scanned-in Data field has been successfully written into the rDTR at the Update-DR state.

13.7.6 Target to host debug communications channel sequence

The DBGTAP debugger can use the following sequence for receiving data from the core:

1. Scan_N into the IR.
2. 5 into the SCREG.
3. INTEST into the IR.
4. Scan out 34 bits of data. If the Valid flag is clear repeat this step again.

5. The least significant 32 bits hold valid data.
6. Go to step 4 again for reading out more data.

13.7.7 Host to target debug communications channel

The DBGTAP debugger can use the following sequence for sending data to the core:

1. Scan_N into the IR.
2. 5 into the SCREG.
3. EXTEST into the IR.
4. Scan in 34 bits, the least significant 32 holding the word to be sent. At the same time, 34 bits were scanned out. If the nRetry flag is clear repeat this step again.
5. Now the data has been written into the rDTR. Go to step 4 again for sending in more data.

13.7.8 Transferring data in debug state

When the core is in debug state, the DBGTAP debugger can transfer data in and out of the core using the instruction execution facilities described in *Executing instructions in debug state* on page 13-23 in addition to scan chain 5. You must ensure that the DSCR[13] execute ARM instruction enable bit is set for the instruction execution mechanism to work. When it is set, the interface for the DBGTAP debugger consists of the following:

- Scan chain 4 (see *Scan chain 4, Instruction Transfer Register (ITR)* on page 13-13). It is used for loading an instruction and for monitoring the status of the execution:

ITR	When the DBGTAPSM goes through the Update-DR state with EXTEST and scan chain 4 selected, and the Ready flag set, the ITR is loaded with the least significant 32 bits of the scan chain.
InstCompl flag	When clear, this flag indicates to the DBGTAP debugger that the last issued instruction has not yet completed execution. While Ready (captured version of InstCompl) is clear, no updates of the ITR and the rDTR occur and the instruction execution mechanism is disabled. No instruction is issued when going through Run-Test/Idle.

- Scan chain 5 (see *Scan chain 5* on page 13-15). It is used for writing in or reading out the data and for monitoring the state of the execution:

  rDTR When the DBGTAPSM goes through the Update-DR state with EXTEST and scan chain 5 selected, and the Ready flag set, the contents of the Data field are loaded into the rDTR.

  wDTR When the DBGTAPSM goes through the Capture-DR state with INTEST or EXTEST selected, the contents of the wDTR are loaded into the Data field of the scan chain.

  InstCompl flag When clear, this flag indicates to the DBGTAP debugger that the last issued instruction has not yet completed execution. While Ready (captured version of InstCompl) is clear, no updates of the ITR and the rDTR occur and the instruction execution mechanism is disabled. No instruction is issued when going through Run-Test/Idle.

- Some flags and control bits at CP14 debug register c1 (DSCR):

  DSCR[13] Execute ARM instruction enable bit. This bit must be set for the instruction execution mechanism to work.

  Sticky precise Data Abort flag
  DSCR[6]. When set, this flag indicates to the DBGTAP debugger that a precise Data Abort occurred while executing an instruction in debug state. While this bit is set, the instruction execution mechanism is disabled. When this flag is set InstCompl stays HIGH, and additional attempts to execute an instruction appear to succeed but do not execute.

  Sticky imprecise Data Abort flag
  DSCR[7]. When set, this flag indicates to the DBGTAP debugger that an imprecise Data Abort occurred while executing an instruction in debug state. This flag does not disable the debug state instruction execution.

13.7.9 Example sequences

This section includes some example sequences to illustrate how to transfer data between the DBGTAP debugger and the core when it is in debug state. The examples are related to accessing the processor memory.

Target to host transfer

The DBGTAP debugger can use the following sequence for reading data from the processor memory system. The sequence assumes that the ARM register r0 contains a pointer to the address of memory at which the read has to start:

1. Scan_N into the IR.
2. 1 into the SCREG.
3. INTEST into the IR.
4. Scan out the contents of the DSCR. This clears the sticky precise Data Abort and sticky imprecise Data Abort flags.
5. Scan_N into the IR.
6. 4 into the SCREG.
7. EXTEST into the IR.
8. Scan in the `LDC p14,c5,[R0],#4` instruction into the ITR.
9. Scan_N into the IR.
10. 5 into the SCREG.
11. INTEST into the IR.
12. Go through Run-Test/Idle state. The instruction loaded into the ITR is issued to the processor pipeline.
13. Scan out 34 bits of data. If the Ready flag is clear repeat this step again.
14. The instruction has completed execution. Store the least significant 32 bits.
15. Go to step 12 again for reading out more data.
16. Scan_N into the IR.
17. 1 into the SCREG.
18. INTEST into the IR.
19. Scan out the contents of the DSCR. This clears the sticky precise Data Abort and sticky imprecise Data Abort flags. If the sticky precise Data Abort is set, this means that during the sequence one of the instructions caused a precise Data Abort. All the instructions that follow are not executed. Register r0 points to the next word to be read, and after the cause for the abort has been fixed the sequence resumes at step 5.

Note

If the sticky imprecise Data Aborts flag is set, an imprecise Data Abort has occurred and the sequence restarts at step 1 after the cause of the abort is fixed and r0 is reloaded.

Host to target transfer

The DBGTAP debugger can use the following sequence for writing data to the processor memory system. The sequence assumes that the ARM register r0 contains a pointer to the address of memory at which the write has to start:

1. Scan_N into the IR.
2. 1 into the SCREG.
3. INTEST into the IR.
4. Scan out the contents of the DSCR. This clears the sticky precise Data Abort and sticky imprecise Data Abort flags.
5. Scan_N into the IR.
6. 4 into the SCREG.
7. EXTEST into the IR.
8. Scan in the `STC p14,c5,[R0],#4` instruction into the ITR.
9. Scan_N into the IR.
10. 5 into the SCREG.
11. EXTEST into the IR.
12. Scan in 34 bits, the least significant 32 holding the word to be sent. At the same time, 34 bits are scanned out. If the Ready flag is clear, repeat this step.
13. Go through Run-Test/Idle state.
14. Go to step 12 again for writing in more data.
15. Scan in 34 bits. All the values are don’t care. At the same time, 34 bits are scanned out. If the Ready flag is clear, repeat this step. The don’t care value is written into the rDTR (Update-DR state) just after Ready is seen set (Capture-DR state). However, the STC instruction is not re-issued because the DBGTAPSM does not go through Run-Test/Idle.

16. Scan_N into the IR.
17. 1 into the SCREG.
18. INTEST into the IR.
19. Scan out the contents of the DSCR. This clears the sticky precise Data Abort and sticky imprecise Data Abort flags. If the sticky precise Data Abort is set, this means that during the sequence one of the instructions caused a precise Data Abort. All the instructions that follow are not executed. Register r0 points to the next word to be written, and after the cause for the abort has been fixed the sequences resumes at step 5.

    Note

    If the sticky imprecise Data Abort flag is set, an imprecise Data Abort has occurred and the sequence restarts at step 1 after the cause of the abort is fixed and c0 is reloaded.

13.8 Debug sequences

This section describes some sequences of operations that a debugger might execute as part of the debugging process. The purpose of this section is to show how the debug features work by providing a hypothetical usage model. A developer of a debugger for the ARM11 MPCore processor must not use the information provided in this section as a recommended implementation.

In Halt mode, the processor stops when a debug event occurs enabling the DBGTAP debugger to do the following:

1. Determine and modify the current state of the processor and memory.
2. Set up breakpoints, watchpoints, and vector catches.
3. Restart the processor.

You enable this mode by setting CP14 debug DSCR[14] bit, which can only be done by the DBGTAP debugger.

From here it is assumed that the debug unit is in Halt mode. Monitor debug-mode debugging is described in *Monitor debug-mode debugging* on page 13-49.

13.8.1 Debug macros

The debug code sequences in this section are written using a fixed set of macros. The mapping of each macro into a debug scan chain sequence is given in this section.

Scan_N <n>

Select scan chain register number <n>:

1. Scan the Scan_N instruction into the IR.
2. Scan the number <n> into the DR.

INTEST

1. Scan the INTEST instruction into the IR.

EXTEST

1. Scan the EXTEST instruction into the IR.

ITRsel

1. Scan the ITRsel instruction into the IR.

Restart

1. Scan the Restart instruction into the IR.
2. Go to the DBGTAP controller Run-Test/Idle state so that the processor exits debug state.

INST <instr> [stateout]

Go through Capture-DR, go to Shift-DR, scan in an ARM instruction to be read and executed by the core and scan out the Ready flag, go through Update-DR. The ITR (scan chain 4) and EXTEST must be selected when using this macro.

1. Scan in:
   - Any value for the InstCompl flag. This bit is read-only.
   - 32-bit assembled code of the instruction (instr) to be executed, for ITR[31:0].
2. The following data is scanned out:
   - The value of the Ready flag, to be stored in stateout.
   - 32 bits to be ignored. The ITR is write-only.

DATA <datain> [<stateout> [dataout]]

Go through Capture-DR, go to Shift-DR. Scan in a data item and scan out another one, go through Update-DR. Either the DTR (scan chain 5) or the DSCR (scan chain 1) must be selected when using this macro.

1. If scan chain 5 is selected, scan in:
   - Any value for the nRetry or Valid flag. These bits are read-only.
   - Any value for the InstCompl flag. This bit is read-only.
   - 32-bit datain value for rDTR[31:0].
2. The following data is scanned out:
   - The contents of wDTR[31:0], to be stored in dataout.
   - If the DSCR[13] execute ARM instruction enable bit is set, the value of the Ready flag is stored in stateout.

- If the DSCR[13] execute ARM instruction enable bit is clear, the nRetry or Valid flag (depending on whether EXTEST or INTEST is selected) is stored in stateout.

3. If scan chain 1 is selected, scan in:
   - 32-bit datain value for DSCR[31:0].

   Stateout and dataout fields are not used in this case.

DATAOUT <dataout>

1. Scan out a data value. DSCR (scan chain 1) and INTEST must be selected when using this macro.
2. If scan chain 1 is selected, scan out the contents of the DSCR, to be stored in dataout.
3. The scanned-in value is discarded, because INTEST is selected.

REQ <address> <data> <nR/W> [<stateout> [dataout]]

Go through Capture-DR, go to Shift-DR, scan in a request and scan out the result of the former one, go through Update-DR. Scan chain 7, and either INTEST or EXTEST, must be selected when using this macro.

1. Scan in:
   - 7-bit address value for Address[6:0]
   - 32-bit data value for Data[31:0]
   - 1-bit nR/W value (0 for read and 1 for write) for the Ready/nRW field.
2. Scan out:
   - the value of the Ready/nRW bit, to be stored in stateout
   - the contents of the Data field, to be stored in dataout.

RTI

1. Go through Run-Test/Idle DBGTAPSM state. This forces the execution of the instruction currently loaded into the ITR, provided the execute ARM instruction enable bit (DSCR[13]) is set, the Ready flag was captured as set, and the sticky precise Data Abort flag is cleared.

13.8.2 General setup

You must setup the following control bits before DBGTAP debugging can take place:

- DSCR[14] Halt/Monitor debug-mode bit must be set to 1. It resets to 0 on power-up.
- DSCR[6] sticky precise Data Abort flag must be cleared down, so that aborts are not detected incorrectly immediately after startup.

The DSCR must be read, the DSCR[14] bit set, and the new value written back. The action of reading the DSCR automatically clears the DSCR[6] sticky precise Data Abort flag.

All individual breakpoints, watchpoints, and vector catches reset disabled on power-up.

13.8.3 Forcing the processor to halt

Scan the Halt instruction into the DBGTAP controller IR and go through Run-Test/Idle.

13.8.4 Entering debug state

To enter debug state you must:

1. Check whether the core has entered debug state, as follows:

```
SCAN_N1         ; select DSCR
INTEST
LOOP
    DATAOUT readDSCR
UNTILreadDSCR[0]==1; until Core Halted bit is set
```

2. Save DSCR, as follows:

```
DATAOUT readDSCR
Save value in readDSCR
```

3. Save wDTR (in case it contains some data), as follows:

```
SCAN_N  5                       ; select DTR
INTEST
DATA    0x00000000 Valid wDTR
If Valid==1 then Save value in wDTR
```

4. Set the DSCR[13] execute ARM instruction enable bit, so instructions can be issued to the core from now:

```
SCAN_N  1                       ; select DSCR
EXTEST
DATA modifiedDSCR               ; modifiedDSCR equals readDSCR with bit
                                ; DSCR[13] set
```

5. Before executing any instruction in debug state you have to drain the write buffer. This ensures that no imprecise Data Aborts can return at a later point:

```
SCAN_N   4                         ; select ITR
EXTEST
INST     MCR p15,0,Rd,c7,c10,4     ; data synchronization barrier
LOOP
    LOOP
        SCAN_N 4                   ; select ITR
        EXTEST
        RTI
        INST 0x0 Ready
   Until Ready == 1
   SCAN_N 1
   INTEST
   DATAOUT readDSCR
Until readDSCR[7]==0
SCAN_N 4
EXTEST
INST NOP                           ;NOP takes the
RTI                                ;imprecise Data Aborts
LOOP
    INST 0 Ready
Until Ready == 1
SCAN_N 1
INTEST
DATAOUT readDSCR                   ;clears DSCR[7]
```

Note

If there is a lingering imprecise Data Abort at the time of executing a Data Synchronization Barrier, the ARM architecture does not define it if this instruction completes successfully, or if it is cancelled by this abort. Therefore, this sequence issues the Data Synchronization Barrier repeatedly until it completes successfully. An additional NOP instruction is inserted at the end of the Data Synchronization Barrier sequence in case the device behavior is to recognize the imprecise Data Abort after this Data Synchronization Barrier instruction completes.

6. Store out r0. It is going to be used to save the rDTR. Use the standard sequence of *Reading a current mode ARM register in the range r0-r14* on page 13-40. Scan chain 5 and INTEST are now selected.

7. Save the rDTR and the rDTRempty bit in three steps:

   a. The rDTRempty bit is the inverted version of DSCR[30] (saved in step 2). If DSCR[30] is clear (register empty) there is no requirement to read the rDTR, go to 7.

b. Transfer the contents of rDTR to r0:

```
ITRSEL                              ; select the ITR and EXTEST
INST    MRC p14,0,R0,c0,c5,0        ; instruction to copy CP14's debug
                                    ; register c5 into R0
RTI
LOOP
    INST 0x00000000 Ready
UNTIL   Ready==1                    ; wait until the instruction ends
```

c. Read r0 using the standard sequence of *Reading a current mode ARM register in the range r0-r14* on page 13-40.

8. Store out CPSR using the standard sequence of *Reading the CPSR/SPSR* on page 13-41.

9. Store out PC using the standard sequence of *Reading the PC* on page 13-41.

10. Adjust the PC to enable you to resume execution later:

    - subtract `0x8` from the stored value if the processor was in ARM state when entering debug state
    - subtract `0x4` from the stored value if the processor was in Thumb state when entering debug state
    - subtract `0x0` from the stored value if the processor was in Java state when entering debug state.

    These values are not dependent on the debug state entry method, (see *Behavior of the PC in debug state* on page 12-34). The entry state can be determined by examining the T and J bits of the CPSR.

11. Cache and MMU preservation measures must also be taken here. This includes saving all the relevant CP15 registers using the standard coprocessor register reading sequence described in *Coprocessor register reads and writes* on page 13-46.

13.8.5 Leaving debug state

To leave debug state:

1. Restore standard ARM registers for all modes, except r0, PC, and CPSR.

2. Cache and MMU restoration must be done here. This includes writing the saved registers back to CP15.

3. Ensure that rDTR and wDTR are empty:

```
ITRSEL                              ; select the ITR and EXTEST
INST    MCR p14,0,R0,c0,c5,0        ; instruction to copy R0 into
                                    ; CP14 debug register c5
```

```
RTI
LOOP
    INST 0x00000000 Ready
UNTIL   Ready==1                ; wait until the instruction ends
SCAN_N 5
INTEST
DATA    0x0 Valid wDTR
```

4. If the wDTR did not contain any valid data on debug state entry go to step 5. Otherwise, restore wDTRfull and wDTR (uses r0 as a temporary register) in two steps.

   a. Load the saved wDTR contents into r0 using the standard sequence of *Writing a current mode ARM register in the range r0-r14* on page 13-40. Now scan chain 5 and EXTEST are selected

   b. Transfer r0 into wDTR:

   ```
   ITRSEL                          ; select the ITR and EXTEST
   INST    MCR p14,0,R0,c0,c5,0    ; instruction to copy R0 into
                                   ; CP14 debug register c5
   RTI
   LOOP
       INST 0x00000000 Ready
   UNTIL   Ready==1                ; wait until the instruction ends
   ```

5. Restore CPSR using the standard CPSR writing sequence described in *Writing the CPSR/SPSR* on page 13-41.

6. Restore the PC using the standard sequence of *Writing the PC* on page 13-42.

7. Restore r0 using the standard sequence of *Writing a current mode ARM register in the range r0-r14* on page 13-40. Now scan chain 5 and EXTEST are selected.

8. Restore the DSCR with the DSCR[13] execute ARM instruction enable bit clear, so no more instructions can be issued to the core:

   ```
   SCAN_N  1                       ; select DSCR
   EXTEST
   DATA modifiedDSCR               ; modifiedDSCR equals the saved contents
                                   ; of the DSCR with bit DSCR[13] clear
   ```

9. If the rDTR did not contain any valid data on debug state entry go to step 10. Otherwise, restore the rDTR and rDTRempty flag:

   ```
   SCAN_N  5                       ; select DTR
   EXTEST
   DATA    Saved_rDTR              ; rDTRempty bit is automatically cleared
                                   ; as a result of this action
   ```

10. Restart processor:

    ```
    RESTART
    ```

11. Wait until the core is restarted:

```
SCAN_N  1                     ; select DSCR
INTEST
LOOP
    DATAOUT readDSCR
UNTIL   readDSCR[1]==1        ; until Core Restarted bit is set
```

13.8.6 Reading a current mode ARM register in the range r0-r14

Use the following sequence to read a current mode ARM register in the range r0-r14:

```
SCAN_N  5                         ; select DTR
ITRSEL                            ; select the ITR and EXTEST
INST    MCR p14,0,Rd,c0,c5,0      ; instruction to copy Rd into CP14 debug
                                  ; register c5
RTI
INTEST                            ; select the DTR and INTEST
LOOP
    DATA 0x00000000 Ready readData
UNTIL   Ready==1                  ; wait until the instruction ends
Save value in readData
```

Note

Register r15 cannot be read in this way because the effect of the required MCR is to take an Undefined exception.

13.8.7 Writing a current mode ARM register in the range r0-r14

Use the following sequence to write a current mode ARM register in the range r0-r14:

```
SCAN_N  5                                 ; select DTR
ITRSEL                                    ; select the ITR and EXTEST
INST    MRC p14,0,Rd,c0,c5,0              ; instruction to copy CP14 debug
                                          ; register c5 into Rd
EXTEST                                    ; select the DTR and EXTEST
DATA    Data2Write
RTI
LOOP
    DATA 0x00000000 Ready
UNTIL   Ready==1                          ; wait until the instruction ends
```

Note

Register r15 cannot be written in this way because the MRC instruction used would update the CPSR flags rather than the PC.

13.8.8 Reading the CPSR/SPSR

Here r0 is used as a temporary register:

1. Move the contents of CPSR/SPSR to r0.

   ```
   SCAN_N  5                        ; select DTR
   ITRSEL                     ; select the ITR and EXTEST
   INST    MRS R0,CPSR              ; or SPSR
   RTI
   LOOP
       INST 0x00000000 Ready
   UNTIL   Ready==1                 ; wait until the instruction ends
   ```

2. Perform the read of r0 using the standard sequence described in *Reading a current mode ARM register in the range r0-r14* on page 13-40. Scan chain 5 and ITRsel are already selected.

13.8.9 Writing the CPSR/SPSR

Here r0 is used as a temporary register:

1. Load the desired value into r0 using the standard sequence described in *Writing a current mode ARM register in the range r0-r14* on page 13-40. Now scan chain 5 and EXTEST are selected.

2. Move the contents of r0 to CPRS/SPRS:

   ```
   ITRSEL                            ; select the ITR and EXTEST
   INST    MSR CPSR,R0              ; or SPSR
   RTI
   LOOP
       INST 0x00000000 Ready
   UNTIL   Ready==1                 ; wait until the instruction ends
   ```

It is not a problem to write to the T and J bits because they have no effect in the execution of instructions while in Debug state.

The CPSR mode and control bits can be written in User mode when the core is in debug state. This is essential so that the debugger can change mode and then get at the other banked registers.

13.8.10 Reading the PC

Here r0 is used as a temporary register:

1. Move the contents of the PC to r0:

   ```
   ITRSEL                           ; select the ITR and EXTEST
   INST    MOV R0,PC
   ```

```
RTI
LOOP
    INST 0x00000000 Ready
UNTIL   Ready==1                ; wait until the instruction ends
```

2. Read the contents of r0 using the standard sequence described in *Reading a current mode ARM register in the range r0-r14* on page 13-40.

13.8.11 Writing the PC

Here r0 is used as a temporary register:

1. Load r0 with the address to resume using the standard sequence described in *Writing a current mode ARM register in the range r0-r14* on page 13-40. Now scan chain 5 and EXTEST are selected.

2. Move the contents of r0 to the PC:

```
ITRSEL                          ; select the ITR and EXTEST
INST    MOV PC,R0
RTI
LOOP
    INST 0x00000000 Ready
UNTIL   Ready==1                ; wait until the instruction ends
```

13.8.12 General notes about reading and writing memory

On the ARM11 MPCore processor, an abort occurring in debug state causes an Abort exception entry sequence to start, and so changes mode to Abort mode, and writes to r14_abt and SPSR_abt. This means that the Abort mode registers must be saved before performing a debug state memory access.

The word-based read and write sequences are substantially more efficient than the halfword and byte sequences. This is because the ARM LDC and STC instructions always perform word accesses, and this can be used for efficient access to word width memory. Halfword and byte accesses must be done with a combination of loads or stores, and coprocessor register transfers, which is much less efficient.

When writing data, the Instruction Cache might become incoherent. In those cases, either a line or the whole Instruction Cache must be invalidated. In particular, the Instruction Cache must be invalidated before setting a software breakpoint or downloading code.

13.8.13 Reading memory as words

This sequence is optimized for a long sequential read.

This sequence assumes that r0 has been set to the address to load data from prior to running this sequence. r0 is post-incremented so that it can be used by successive reads of memory.

1. Load and issue the LDC instruction:

   ```
   SCAN_N  5                        ; select DTR
   ITRSEL                           ; select the ITR and EXTEST
   INST    LDC p14,c5,[R0],#4       ; load the content of the position of
                                    ; memory pointed by R0 into wDTR and
                                    ; increment R0 by 4
   RTI
   ```

2. The DTR is selected in order to read the data:

   ```
   INTEST                           ; select the DTR and INTEST
   ```

3. This loop keeps on reading words, but it stops before the latest read. It is skipped if there is only one word to read:

   ```
   FOR(i=1; i <= (Words2Read-1); i++) DO
           LOOP
                   DATA 0x00000000 Ready readData   ; gets the result of
                                                    ; the previous read
                   RTI             ; issues the next read
           UNTIL Ready==1          ; wait until the instruction ends
           Save value in readData
   ENDFOR
   ```

4. Wait for the last read to finish:

   ```
   LOOP
       DATA 0x00000000 Ready readData
   UNTIL Ready==1          ; wait until instruction ends
   Save value in readData
   ```

5. Now check whether an abort occurred:

   ```
   SCAN_N  1                        ; select DSCR
   INTEST
   DATAOUT DSCR                     ; this action clears the DSCR[6] flag
   ```

6. Scan out the contents of the DSCR. This clears the sticky precise Data Abort and sticky imprecise Data Abort flags. If the sticky precise Data Abort is set, this means that during the sequence one of the instructions caused a precise Data Abort. All the instructions that follow are not executed. Register r0 points to the next word to be written, and after the cause for the abort has been fixed the sequences resumes at step 1.

Note

If the sticky imprecise Data Aborts flag is set, an imprecise Data Abort has occurred and the sequence restarts at step 1 after the cause of the abort is fixed and r0 is reloaded.

13.8.14 Writing memory as words

This sequence is optimized for a long sequential write.

This sequence assumes that r0 has been set to the address to store data to prior to running this sequence. Register r0 is post-incremented so that it can be used by successive writes to memory:

1. The instruction is loaded:

```
SCAN_N   5                         ; select DTR
ITRSEL                             ; select the ITR and EXTEST
INST     STC p14,c5,[R0],#4        ; store the contents of rDTR into the
                                   ; position of memory pointed by R0 and
                                   ; increment it by 4
EXTEST                             ; select the DTR and EXTEST
```

2. This loop writes all the words:

```
FOR (i=1; i <= Words2Write; i++) DO
        LOOP
                DATA Data2Write Ready
                RTI
        UNTIL Ready==1             ; wait until instruction ends
ENDFOR
```

3. Wait for the last write to finish:

```
LOOP
    DATA 0x00000000 Ready
UNTIL Ready==1                     ; wait until instruction ends
```

4. Check for aborts, as described in *Reading memory as words* on page 13-42.

13.8.15 Reading memory as halfwords or bytes

The above sequences cannot be used to transfer halfwords or bytes because LDC and STC instructions always transfer whole words. Two operations are required to complete a halfword or byte transfer, from memory to ARM register and from ARM register to CP14 debug register. Therefore, performance is decreased because the load instruction cannot be kept in the ITR.

This sequence assumes that r0 has been set to the address to load data from prior to running the sequence. Register r0 is post-incremented so that it can be used by successive reads of memory. Register r1 is used as a temporary register:

1. Load and issue the LDRH or LDRB instruction:

   ```
   ITRSEL                              ; select the ITR and EXTEST
   INST    LDRH R1,[R0],#2             ; LDRB R1,[R0],#1 for byte reads
   RTI
   LOOP
       INST 0x00000000 Ready
   UNTIL Ready==1                      ; wait until instruction ends
   ```

2. Use the standard sequence described in *Reading a current mode ARM register in the range r0-r14* on page 13-40 on register r1. Now scan chain 5 and INTEST are selected.

3. If there are more halfwords or bytes to be read go to 1.

4. Check for aborts, as described in *Reading memory as words* on page 13-42.

13.8.16 Writing memory as halfwords/bytes

This sequence assumes that r0 has been set to the address to store data to prior to running this sequence. Register r0 is post-incremented so that it can be used by successive writes to memory. Register r1 is used as a temporary register:

1. Write the halfword/byte onto r1 using the standard sequence described in *Writing a current mode ARM register in the range r0-r14* on page 13-40. Scan chain 5 and EXTEST are selected.

2. Write the contents of r1 to memory:

   ```
   ITRSEL                              ; select the ITR and EXTEST
   INST    STRH R1,[R0],#2             ; STRB R1,[R0],#1 for byte writes
   RTI
   LOOP
       INST 0x00000000 Ready
   UNTIL Ready==1                      ; wait until instruction ends
   ```

3. If there are more halfwords or bytes to be read go to 1.

4. Now check for aborts as described in *Reading memory as words* on page 13-42.

13.8.17 Coprocessor register reads and writes

The ARM11 MPCore processor can execute coprocessor instructions while in debug state. Therefore, the straightforward method to transfer data between a coprocessor and the DBGTAP debugger is using an ARM register temporarily. For this method to work, the coprocessor must be able to transfer all its registers to the core using coprocessor transfer instructions.

13.8.18 Reading coprocessor registers

1. Load the value into ARM register r0:

   ```
   ITRSEL                              ; select the ITR and EXTEST
   INST    MRC px,y,R0,ca,cb,z
   RTI
   LOOP
       INST 0x00000000 Ready
   UNTIL Ready==1                      ; wait until instruction ends
   ```

2. Use the standard sequence described in *Reading a current mode ARM register in the range r0-r14* on page 13-40.

13.8.19 Writing coprocessor registers

1. Write the value onto r0, using the standard sequence. See *Writing a current mode ARM register in the range r0-r14* on page 13-40 for more details. Scan chain 5 and EXTEST are selected.

2. Transfer the contents of r0 to a coprocessor register:

   ```
   ITRSEL                              ; select the ITR and EXTEST
   INST    MCR px,y,R0,ca,cb,z
   RTI
   LOOP
       INST 0x00000000 Ready
   UNTIL Ready==1                      ; wait until instruction ends
   ```

13.9 Programming debug events

The following operations are described:

- *Reading registers using scan chain 7*
- *Writing registers using scan chain 7*
- *Setting breakpoints, watchpoints and vector catches* on page 13-48
- *Setting software breakpoints* on page 13-48.

13.9.1 Reading registers using scan chain 7

A typical sequence for reading registers using scan chain 7 is as follows:

```
SCAN_N  7                        ; select ITR
EXTEST
REQ 1stAddr2Rd 0 0               ;read request for register 1stAddr2read
FOR(i=2; i <= Words2Read; i++) DO
        LOOP
                REQ ithAddr2Rd 0 0 Ready readData
                                 ; ith read request while waiting
        UNTIL Ready==1           ; wait until the previous request completes
        Save value in readData
ENDFOR
LOOP
        REQ 0 0 0 Ready readData ; null request while waiting
UNTIL Ready==1                   ; wait until last request completes
Save value in readData
```

13.9.2 Writing registers using scan chain 7

A typical sequence for writing to a register using scan chain 7 is as follows:

```
SCAN_N  7                        ; select ITR
EXTEST
REQ 1stAddr2Wr 1stData2Wr 0b1    ; write request for register 1stAddr2write
FOR(i=2; i <= Words2Write; i++) DO
        LOOP
                REQ ithAddr2Wr ithData2Wr 1 Ready
                                 ; ith write request while waiting
        UNTIL Ready==1           ; wait until the previous request completes
ENDFOR
LOOP
        REQ 0 0 0 Ready          ; null request while waiting
UNTIL Ready==1                   ; wait until last request completes
```

13.9.3 Setting breakpoints, watchpoints and vector catches

You can program a vector catch debug event by writing to CP14 Debug Vector Catch Register.

You can program a breakpoint debug event by writing to CP14 debug 64-69 Breakpoint Value Registers and CP14 debug 80-84 Breakpoint Control Registers.

You can program a watchpoint debug event by writing to CP14 debug 96-97 Watchpoint Value Registers and CP14 debug 112-113 Watchpoint Control Registers.

Note

An external debugger can access the CP14 debug registers whether the processor is in debug state or not, so these debug events can be programmed on-the-fly (while the processor is in ARM, Thumb or Java state).

See *Setting breakpoints, watchpoints, and vector catch debug events* on page 12-39 for the sequences of register accesses to program these software debug events. See *Writing registers using scan chain 7* on page 13-47 to learn how to access CP14 debug registers using scan chain 7.

13.9.4 Setting software breakpoints

To set a software breakpoint on a certain Virtual Address, a debugger must go through the following steps:

1. Read memory location and save the actual instruction.
2. Write the BKPT instruction to the memory location.
3. Read memory location again to check that the BKPT instruction was written.
4. If it is not written, determine the reason.

All of these can be done using the previously described sequences.

Note

Cache coherency issues might arise when writing a BKPT instruction.

13.10 Monitor debug-mode debugging

If DSCR[14] Halt or Monitor debug-mode bit is clear, then the processor takes an exception (rather than halting) when a software debug event occurs. See *Halt mode debugging* on page 12-45 for details.

When the exception is taken, the handler uses the DCC to transmit status information to, and receive commands from the host using a DBGTAP debugger. Monitor debug-mode is essential in real-time systems when the core cannot be halted to collect information.

13.10.1 Receiving data from the core

```
SCAN_N  5                       ; select DTR
INTEST
FOREACH Data2Read
        LOOP
                DATA 0x00000000 Valid readData
        UNTIL Valid==1          ; wait until instruction ends
        Save value in readData
END
```

13.10.2 Sending data to the core

```
SCAN_N  5                       ; select DTR
EXTEST
FOREACH Data2Write
        LOOP
                DATA Data2Write nRetry
        UNTIL nRetry==1     ; wait until instruction ends
END
```

Chapter 14
Cycle Timings and Interlock Behavior

This chapter describes the cycle timings and interlock behavior of integer instructions on MP11 CPUs. It contains the following sections:

- *Thumb instructions* on page 14-32.

14.1 About cycle timings and interlock behavior

Complex instruction dependencies and memory system interactions make it impossible to describe briefly the exact cycle timing behavior for all instructions in all circumstances. The timings described in this chapter are accurate in most cases. If precise timings are required you must use a cycle-accurate model of the MP11 CPUs.

Unless stated otherwise cycle counts and result latencies described in this chapter are best case numbers. They assume:

- no outstanding data dependencies between the current instruction and a previous instruction
- the instruction does not encounter any resource conflicts
- all data accesses hit in the MicroTLB and Data Cache, and do not cross protection region boundaries
- all instruction accesses hit in the Instruction Cache.

This section describes:

- *Changes in instruction flow overview*
- *Instruction execution overview* on page 14-5
- *Conditional instructions* on page 14-6
- *Opposite condition code checks* on page 14-7
- *Definition of terms* on page 14-4.

14.1.1 Changes in instruction flow overview

To minimize the number of cycles, because of changes in instruction flow, each MP11 CPU includes a:

- dynamic branch predictor
- static branch predictor
- return stack.

The dynamic branch predictor is a 128-entry direct-mapped branch predictor using VA bits [9:3]. The prediction scheme uses a two-bit saturating counter for predictions that are:

- Strongly Not Taken
- Weakly Not Taken
- Weakly Taken
- Strongly Taken.

Only branches with a constant offset are predicted. Branches with a register-based offset are not predicted. A dynamically predicted branch can be folded out of the instruction stream if the following instruction arrives while the branch is within the prefetch instruction buffer. A dynamically predicted branch takes one cycle or zero cycles if folded out.

The static branch predictor operates on branches with a constant offset that are not predicted by the dynamic branch predictor. Static predictions are issued from the Iss stage of the main pipeline, consequently a statically predicted branch takes four cycles.

The return stack consists of three entries, and as with static predictions, issues a prediction from the Iss stage of the main pipeline. The return stack mispredicts if the value taken from the return stack is not the value that is returned by the instruction. Only unconditional returns are predicted. A conditional return pops an entry from the return stack but is not predicted. If the return stack is empty a return is not predicted. Items are placed on the return stack from the following instructions:

- `BL #<immed>`
- `BLX #<immed>`
- `BLX Rx`

Items are popped from the return stack by the following types of instruction:

- `BX lr`
- `MOV pc, lr`
- `LDR pc, [sp], #cns`
- `LDMIA sp!, {….,pc}`

A correctly predicted return stack pop takes four cycles.

14.1.2 Definition of terms

Table 14-1 gives descriptions of cycle timing terms used in this chapter.

Table 14-1 Definition of cycle timing terms

Term	Description
Cycles	This is the minimum number of cycles required by an instruction.
Result latency	This is the number of cycles before the result of this instruction is available for a following instruction requiring the result at the start of the ALU, MAC2, and DC1 stage. This is the normal case. Exceptions to this mark the register as an Early Reg. **Note** The result latency is the number of cycles from the first cycle of an instruction.

Table 14-1 Definition of cycle timing terms (continued)

Term	Description
Register lock latency	For STM and STRD instructions only. This is the number of cycles that a register is write locked for by this instruction, preventing subsequent instructions that want to write the register from starting. This lock is required to prevent a following instruction from writing to a register before it has been read.
Early Reg	The specified registers are required at the start of the Sh, MAC1, and ADD stage. Add one cycle to the result latency of the instruction producing this register for interlock calculations.
Late Reg	The specified registers are not required until the start of the ALU, MAC2, and DC1 stage for the second execution. Subtract one cycle from the result latency of the instruction producing this register for interlock calculations.
FlagsCycleDistance	The number of cycles between an instruction that sets the flags and the conditional instruction.

14.1.3 Instruction execution overview

The instruction execution pipeline is constructed from three parallel four-stage pipelines, see Table 14-2. For a complete description of these pipeline stages see *Pipeline stages* on page 1-22.

Table 14-2 Pipeline stages

Pipeline	Stages			
ALU	Sh	ALU	Sat	WBex
Multiply	MAC1	MAC2	MAC3	
Load/Store	ADD	DC1	DC2	WBls

The ALU and multiply pipelines operate in a lock-step manner, causing all instructions in these pipelines to retire in order. The load/store pipeline is a decoupled pipeline enabling subsequent instructions in the ALU and multiply pipeline to complete underneath outstanding loads.

Extensive forwarding to the Sh, MAC1, ADD, ALU, MAC2, and DC1 stages enables many dependent instruction sequences to run without pipeline stalls. General forwarding occurs from the ALU, Sat, WBex and WBls pipeline stages. In addition, the multiplier contains an internal multiply accumulate forwarding path.

Most instructions do not require a register until the ALU stage. All result latencies are given as the number of cycles until the register is required by a following instruction in the ALU stage.

The following sequence takes four cycles:

```
LDR r1, [r2]              ;Result latency three
ADD r3, r3, r1            ;Register r1 required by ALU
```

If a subsequent instruction requires the register at the start of the Sh, MAC1, or ADD stage then an extra cycle must be added to the result latency of the instruction producing the required register. Instructions that require a register at the start of these stages are specified by describing that register as an Early Reg. The following sequence, requiring an Early Reg, takes five cycles:

```
LDR r1, [r2]              ;Result latency three plus one
ADD r3, r3, r1 LSL#6      ;plus one since Register r1 is required by Sh
```

Finally, some instructions do not require a register until their second execution cycle. If a register is not required until the ALU, MAC1, or DC1 stage for the second execution cycle, then a cycle can be subtracted from the result latency for the instruction producing the required register. If a register is not required until this later point, it is specified as a Late Reg. The following sequence where r1 is a Late Reg takes four cycles:

```
LDR r1, [r2]              ;Result latency three minus one
ADD r3, r1, r3, LSLr4     ;minus one since Register r1 is a Late Reg
                          ;This ADD is a two issue cycle instruction
```

14.1.4 Conditional instructions

Most instructions execute in one or two cycles. If these instructions fail their condition codes then they take one and two cycles respectively.

Multiplies, MSR, and some CP14 and CP15 coprocessor instructions are the only instructions that require more than two cycles to execute. If one of these instructions fails its condition codes, then it takes a variable number of cycles to execute. The number of cycles is dependent on:

- the length of the operation
- the number of cycles between the setting of the flags and the start of the dependent instruction.

The worst-case number of cycles for a condition code failing multicycle instruction is five.

The following algorithm describes the number of cycles taken for multi-cycle instructions which condition code fail:

Min(NonFailingCycleCount, Max(5 - FlagCycleDistance, 3))

Where:

Max (a,b)	returns the maximum of the two values a,b.
Min (a,b)	returns the minimum of the two values a,b.
NonFailingCycleCount	is the number of cycles that the failing instruction would have taken had it passed.

FlagCycDistance is the number of cycles between the instruction that sets the flags and the conditional instruction, including interlocking cycles. For example:

- The following sequence has a FlagCycleDistance of 0 because the instructions are back-to-back with no interlocks:

  ```
  ADDS r1, r2, r3
  MULEQ r4, r5, r6
  ```

- The following sequence has a FlagCycleDistance of one:

  ```
  ADDS r1, r2, r3
  MOV   r0, r0
  MULEQ r4, r5, r6
  ```

14.1.5 Opposite condition code checks

If instruction A and instruction B both write the same register the pipeline must ensure that the register is written in the correct order. Therefore interlocks might be required to correctly resolve this pipeline hazard.

The only useful sequences where two instructions write the same register without an instruction reading its value in between are when the two instructions have opposite sets of condition codes. The MP11 CPUs optimize these sequences to prevent unnecessary interlocks. For example:

- The following sequences take two cycles to execute:

  -
    ```
    ADDNE r1, r5, r6
    LDREQ r1, [r8]
    ```

  -
    ```
    LDREQ r1, [r8]
    ADDNE r1, r5, r6
    ```
- The following sequence also takes two cycles to execute, because the STR instruction does not store the value of r1 produced by the QDADDNE instruction:

  ```
  QDADDNE r1, r5, r6
  STREQ r1, [r8]
  ```

14.2 Register interlock examples

Table 14-3 shows register interlock examples using LDR and ADD instructions.

LDR instructions take one cycle, have a result latency of three, and require their base register as an Early Reg.

ADD instructions take one cycle and have a result latency of one.

Table 14-3 Register interlock examples

Instruction sequence	Behavior

`LDR r1, [r2]`
`ADD r6, r5, r4`
	Takes two cycles because there are no register dependencies

`ADD r1, r2, r3`
`ADD r9, r6, r1`
	Takes two cycles because ADD instructions have a result latency of one

`LDR r1, [r2]`
`ADD r6, r5, r1`
	Takes four cycles because of the result latency of r1

`ADD r2, r5, r6`
`LDR r1, [r2]`
	Takes three cycles because of the use of the result of r1 as an Early Reg

`LDR r1, [r2]`
`LDR r5, [r1]`
	Takes five cycles because of the result latency and the use of the result of r1 as an Early Reg

14.3 Data processing instructions

This section describes the cycle timing behavior for the AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, CMN, ORR, MOV, BIC, MVN, TST, TEQ, CMP, and CLZ instructions.

14.3.1 Cycle counts if destination is not the PC

Table 14-4 shows the cycle timing behavior for data processing instructions if the destination is not the PC. You can substitute ADD with any of the data processing instructions identified in the opening paragraph of this section.

Table 14-4 Data Processing instruction cycle timing behavior if destination is not PC

Example instruction	Cycles	Early Reg	Late Reg	Result latency	Comment
`ADD <Rd>, <Rn>, <Rm.`	1	-	-	1	Normal case.
`ADD <Rd>, <Rn>, <Rm>, LSL #<immed>`	1	`<Rm>`	-	1	Requires a shifted source register.
`ADD <Rd>, <Rn>, <Rm>, LSL <Rs>`	2	`<Rs>`	`<Rn>`	2	Requires a register controlled shifted source register. Instruction takes two issue cycles. In the first cycle the shift distance `Rs` is sampled. In the second cycle the actual shift of `Rm` and the ADD instruction occurs.

14.3.2 Cycle counts if destination is the PC

Table 14-5 shows the cycle timing behavior for data processing instructions if the destination is the PC. You can substitute ADD with any data processing instruction except for a MOV and CLZ. A CLZ with the PC as the destination is an Unpredictable instruction.

The timings for a MOV instruction are given separately in the Table 14-5.

For condition code failing cycle counts, the cycles for the non-PC destination variants must be used.

Table 14-5 Data processing instruction cycle timing behavior if destination is the PC

Example instruction	Cycles	Early Reg	Late Reg	Result latency	Comment
`MOV pc,lr`	4	-	-	-	Correctly predicted return stack

Table 14-5 Data processing instruction cycle timing behavior if destination is the PC (continued)

Example instruction	Cycles	Early Reg	Late Reg	Result latency	Comment
`MOV pc,lr`	7	-	-	-	Incorrectly predicted return stack
`MOV <cond> pc, lr`	5-7[a]	-	-	-	Conditional return, or return when return stack is empty
`MOV pc, <Rd>`	5	-	-	-	MOV to PC, no shift required
`MOV <cond> pc, <Rd>`	5-7[a]	-	-	-	Conditional MOV to PC, no shift required
`MOV pc, <Rn>, <Rm>, LSL #<immed>`	6	<Rm>	-	-	Conditional MOV to PC, with a shifted source register
`MOV <cond> pc, <Rn>, <Rm>, LSL #<immed>`	6-7[a]	-	-	-	Conditional MOV to PC, with a shifted source register
`MOV pc, <Rn>, <Rm>, LSL <Rs>`	7	<Rs>	<Rn>	-	MOV to PC, with a register controlled shifted source register
`ADD pc, <Rd>, <Rm>`	7	-	-	-	Normal case to PC
`ADD pc, <Rn>, <Rm>, LSL #<immed>`	7	<Rm>	-	-	Requires a shifted source register
`ADD pc, <Rn>, <Rm>, LSL <Rs>`	8	<Rs>	<Rn>	-	Requires a register controlled shifted source register

a. If the instruction is conditional and passes conditional checks it takes MAX(MaxCycles - FlagCycleDistance, MinCycles), If the instruction is unconditional it takes Min Cycles.

14.3.3 Example interlocks

Most data processing instructions are single-cycle and can be executed back-to-back without interlock cycles, even if there are data dependencies between them. The exceptions to this are when the shifter or register controlled shifts are used.

Shifter

The shifter is in a separate pipeline stage from the ALU. A register required by the shifter is an Early Reg and requires an additional cycle of result availability before use. For example, the following sequence introduces a one-cycle interlock, and takes three cycles to execute:

```
ADD r1,r2,r3
ADD r4,r5,r1 LSL #1
```

The second source register, which is not shifted, does not incur an extra data dependency check. Therefore, the following sequence takes two cycles to execute:

```
ADD r1,r2,r3
ADD r4,r1,r9 LSL #1
```

Register controlled shifts

Register controlled shifts take two cycles to execute:

- the register containing the shift distance is read in the first cycle
- the shift is performed in the second cycle
- The final operand is not required until the ALU stage for the second cycle.

Because a shift distance is required, the register containing the shift distance is an Early Reg and incurs an extra interlock penalty. For example, the following sequence takes four cycles to execute:

```
ADD r1, r2, r3
ADD r4, r2, r4, LSL r1
```

14.4 QADD, QDADD, QSUB, and QDSUB instructions

This section describes the cycle timing behavior for the QADD, QDADD, QSUB, and QDSUB instructions.

These instructions perform saturating arithmetic. Their result is produced during the Sat stage, consequently they have a result latency of two. The QDADD and QDSUB instructions must double and saturate the register Rn before the addition. This operation occurs in the Sh stage of the pipeline, consequently this register is an Early Reg.

Table 14-6 shows the cycle timing behavior for QADD, QDADD, QSUB, and QDSUB instructions.

Table 14-6 QADD, QDADD, QSUB, and QDSUB instruction cycle timing behavior

Instructions	Cycles	Early Reg	Result latency
QADD, QSUB	1	-	2
QDADD, QDSUB	1	<Rn>	2

14.5 ARMv6 media data processing

Table 14-7 shows ARMv6 media data processing instructions and gives their cycle timing behavior.

All ARMv6 media data processing instructions are single-cycle issue instructions. These instructions produce their results in either the ALU or Sat stage, and have result latencies of one or two accordingly. Some of the instructions require an input register to be shifted before use and therefore are marked as requiring an Early Reg.

Table 14-7 ARMv6 media data processing instructions cycle timing behavior

Instructions	Cycles	Early Reg	Result latency
SADD16, SSUB16, SADD8, SSUB8	1	-	1
USAD8, USADA8	1	<Rm>,<Rs>	3
UADD16, USUB16, UADD8, USUB8	1	-	1
SEL	1	-	1
QADD16, QSUB16, QADD8, QSUB8	1	-	2
SHADD16, SHSUB16, SHADD8, SHSUB8	1	-	2
UQADD16, UQSUB16, UQADD8, UQSUB8	1	-	2
UHADD16, UHSUB16, UHADD8, UHSUB8	1	-	2
SSAT16, USAT16	1	-	2
SADDSUBX, SSUBADDX	1	<Rm>	1
UADDSUBX, USUBADDX	1	<Rm>	1
SADD8TO16, SADD8TO32, SADD16TO32	1	<Rm>	1
SUNPK8TO16, SUNPK8TO32, SUNPK16TO32	1	<Rm>	1
UUNPK8TO16, UUNPK8TO32, UUNPK16TO32	1	<Rm>	1
UADD8TO16, UADD8TO32, UADD16TO32	1	<Rm>	1
REV, REV16, REVSH	1	<Rm>	1
PKHBT, PKHTB	1	<Rm>	1
SSAT, USAT	1	<Rm>	2
QADDSUBX, QSUBADDX	1	<Rm>	2

Table 14-7 ARMv6 media data processing instructions cycle timing behavior

Instructions	Cycles	Early Reg	Result latency
SHADDSUBX, SHSUBADDX	1	<Rm>	2
UQADDSUBX, UQSUBADDX	1	<Rm>	2
UHADDSUBX, UHSUBADDX	1	<Rm>	2

14.6 ARMv6 Sum of Absolute Differences (SAD)

Table 14-8 shows ARMv6 SAD instructions and gives their cycle timing behavior.

Table 14-8 ARMv6 sum of absolute differences instruction timing behavior

Instructions	Cycles	Early Reg	Result latency
USAD8	1	<Rm>,<Rs>	3[a]
USADA8	1	<Rm>,<Rs>	3[a]

a. Result latency is one less If the destination is the accumulate for a subsequent USADA8.

14.6.1 Example interlocks

Table 14-9 shows interlock examples using USAD8 and USADA8 instructions.

Table 14-9 Example interlocks

Instruction sequence	Behavior

`USAD8 r1,r2,r3`
`ADD   r5,r6,r1`
	Takes four cycles because USAD8 has a result latency of three, and the ADD requires the result of the USAD8 instruction.

`USAD8 r1,r2,r3`
`MOV   r9,r9`
`MOV   r9,r9`
`ADD   r5,r6,r1`
	Takes four cycles. The MOV instructions are scheduled during the result latency of the USAD8 instruction.

`USAD8  r1,r2,r3`
`USADA8 r1,r4,r5,r1`
	Takes three cycles. The result latency is one less because the result is used as the accumulate for a subsequent USADA8 instruction.

14.7 Multiplies

The multiplier consists of a three-cycle pipeline with early result forwarding not possible, other than to the internal accumulate path. For a subsequent multiply accumulate the result is available one cycle earlier than for all other uses of the result.

Certain multiplies require:

- more than one cycle to execute.
- more than one pipeline issue to produce a result.

Multiplies with 64-bit results take and require two cycles to write the results, consequently they have two result latencies with the low half of the result always available first. The multiplicand and multiplier are required as Early Regs because they are both required at the start of MAC1.

Table 14-10 shows the cycle timing behavior of example multiply instructions.

Table 14-10 Example multiply instruction cycle timing behavior

Example instruction	Cycles	Cycles if sets flags	Early Reg	Late Reg	Result latency
MUL(S)	2	5	<Rm>, <Rs>	-	4
MLA(S)	2	5	<Rm>, <Rs>	<Rn>	4
SMULL(S)	3	6	<Rm>, <Rs>	-	4/5
UMULL(S)	3	6	<Rm>, <Rs>	-	4/5
SMLAL(S)	3	6	<Rm>, <Rs>	<RdLo>	4/5
UMLAL(S)	3	6	<Rm>, <Rs>	<RdLo>	4/5
SMULxy	1	-	<Rm>, <Rs>	-	3
SMLAxy	1	-	<Rm>, <Rs>	-	3
SMULWy	1	-	<Rm>, <Rs>	-	3
SMLAWy	1	-	<Rm>, <Rs>	-	3
SMLALxy	2	-	<Rm>, <Rs>	<RdHi>	3/4
SMUAD, SMUADX	1	-	<Rm>, <Rs>	-	3
SMLAD, SMLADX	1	-	<Rm>, <Rs>	-	3
SMUSD, SMUSDX	1	-	<Rm>, <Rs>	-	3
SMLSD, SMLSDX	1	-	<Rm>, <Rs>	-	3

Table 14-10 Example multiply instruction cycle timing behavior (continued)

Example instruction	Cycles	Cycles if sets flags	Early Reg	Late Reg	Result latency
SMMUL, SMMULR	2	-	<Rm>, <Rs>	-	4
SMMLA, SMMLAR	2	-	<Rm>, <Rs>	<Rn>	4
SMMLS, SMMLSR	2	-	<Rm>, <Rs>	<Rn>	4
SMLALD, SMLALDX	2	-	<Rm>, <Rs>	<RdHi>	3/4
SMLSLD, SMLSLDX	2	-	<Rm>, <Rs>	<RdHi>	3/4
UMAAL	3	-	<Rm>, <Rs>	<RdLo>	4/5

Note

Result latency is one less if the result is used as the accumulate register for a subsequent multiply accumulate.

14.8 Branches

This section describes the cycle timing behavior for the B, BL, and BLX instructions.

Branches are subject to dynamic, static, and return stack predictions. Table 14-11 shows example branch instructions and their cycle timing behavior.

Table 14-11 Branch instruction cycle timing behavior

Example instruction	Cycles	Comment
`B <immed>`	0	Folded dynamic prediction
`B<immed>, BL<immed>, BLX<immed>`	1	Not-folded dynamic prediction
`B<immed>, BL<immed>, BLX<immed>`	1	Correct not-taken static prediction
`B<immed>, BL<immed>, BLX<immed>`	4	Correct taken static prediction
`B<immed>, BL<immed>, BLX<immed>`	5-7[a]	Incorrect dynamic/static prediction
`BX r14`	4	Correct return stack prediction
`BX r14`	7	Incorrect return stack prediction
`BX r14`	5	Empty return stack
`BX <cond> r14`	5-7[a]	Conditional return
`BX <cond> <reg>`, `BLX <cond> <reg>`	1	If not taken
`BX <cond> <reg>`, `BLX <cond> <reg>`	5-7[a]	If taken

a. Mispredicted branches, including taken unpredicted branches, take a varying number of cycles to execute depending on their distance from a flag setting instruction. The timing behavior is Cycle = MAX(MaxCycles - FlagCycleDistance, MinCycles).

14.9 Processor state updating instructions

This section describes the cycle timing behavior for the MSR, MRS, CPS, and SETEND instructions. Table 14-12 shows processor state updating instructions and their cycle timing behavior.

Table 14-12 Processor state updating instructions cycle timing behavior

Instruction	Cycles	Comments
`MRS`	1	All MRS instructions
`MSR CPSR_f`	1	MSR to CPSR flags only
`MSR`	4	All other MSRs to the CPSR
`MSR SPSR`	5	All MSRs to the SPSR
`CPS <effect> <iflags>`	1	Interrupt masks only
`CPS <effect> <iflags>, #<mode>`	2	Mode changing
`SETEND`	1	-

14.10 Single load and store instructions

This section describes the cycle timing behavior for LDR, LDRT,LDRB, LDRBT, LDRSB, LDRH, LDRSH, STR, STRT, STRB, STRBT, STRH, and PLD instructions.

Table 14-13 shows the cycle timing behavior for stores and loads, other than loads to the PC.

You can replace LDR with any of the above single load or store instructions. The following rules apply:

- They are single-cycle issue if a constant offset is used or if a register offset with no shift, or shift by 2 is used. Both the base and any offset register are Early Regs.
- They are two-cycle issue if either a negative register offset or a shift other than LSL #2 is used. Only the offset register is an Early Reg.
- If ARMv6 unaligned support is enabled then accesses to addresses not aligned to the access size generates two accesses to memory, and so consume the load/store unit for an additional cycle. This extra cycle is required if the base or the offset is not aligned to the access size, consequently the final address is potentially unaligned, even if the final address turns out to be aligned.
- If ARMv6 unaligned support is enabled and the final access address is unaligned there is an extra cycle of result latency.
- PLD (data preload hint instructions) have cycle timing behavior as for load instructions. Because they have no destination register, the result latency is not-applicable for such instructions.
- The updated base register has a result latency of one. For back-to-back load/store instructions with base write back, the updated base is available to the following load/store instruction with a result latency of 0.

Table 14-13 Cycle timing behavior for stores and loads, other than loads to the PC

Example instruction	Cycles	Memory cycles	Result latency	Comments
`LDR <Rd>, <addr_md_1cycle>`[a]	1	1	3	Legacy access / ARMv6 aligned access
`LDR <Rd>, <addr_md_2cycle>`[a]	2	2	4	Legacy access / ARMv6 aligned access
`LDR <Rd>, <addr_md_1cycle>`[a]	1	2	3	Potentially ARMv6 unaligned access

Table 14-13 Cycle timing behavior for stores and loads, other than loads to the PC (continued)

Example instruction	Cycles	Memory cycles	Result latency	Comments
`LDR <Rd>, <addr_md_2cycle>`[a]	2	3	4	Potentially ARMv6 unaligned access
`LDR <Rd>, <addr_md_1cycle>`[a]	1	2	4	ARMv6 unaligned access
`LDR <Rd>, <addr_md_2cycle>`[a]	1	2	4	ARMv6 unaligned access

a. See Table 14-15 on page 14-22 for an explanation of `<addr_md_1cycle>` and `<addr_md_2cycle>`.

Table 14-14 shows the cycle timing behavior for loads to the PC.

Table 14-14 Cycle timing behavior for loads to the PC

Example instruction	Cycles	Memory cycles	Result latency	Comments
`LDR pc, [sp, #cns] (!)`	4	1	-	Correctly return stack predicted
`LDR pc, [sp], #cns`	4	1	-	Correctly return stack predicted
`LDR pc, [sp, #cns] (!)`	9	1	-	Return stack mispredicted
`LDR pc, [sp], #cns`	9	1	-	Return stack mispredicted
`LDR <cond> pc, [sp, #cns] (!)`	8	1	-	Conditional return, or empty return stack
`LDR <cond> pc, [sp], #cns`	8	1	-	Conditional return, or empty return stack
`LDR pc, <addr_md_1cycle>`[a]	8	1	-	-
`LDR pc, <addr_md_2cycle>`[a]	9	2	-	-

a. Table 14-15 on page 14-22 for an explanation of `<addr_md_1cycle>` and `<addr_md_2cycle>`.

Only cycle times for aligned accesses are given because unaligned accesses to the PC are not supported.

The MP11 CPUs include a three-entry return stack that can predict procedure returns. Any load to the PC with an immediate offset, and the stack pointer r13 as the base register is considered a procedure return.

For condition code failing cycle counts, you must use the cycles for the non-PC destination variants.

Table 14-15 on page 14-22 shows the explanation of `<addr_md_1cycle>` and `<addr_md_2cycle>` used in Table 14-13 on page 14-20 and Table 14-14.

Table 14-15 <addr_md_1cycle> and <addr_md_2cycle> LDR example instruction

Example instruction	Early Reg	Comment
`<addr_md_1cycle>`		
`LDR <Rd>, [<Rn>, #cns] (!)`	<Rn>	If an immediate offset, or a positive register offset with no shift or shift LSL #2, then one-issue cycle.
`LDR <Rd>, [<Rn>, <Rm>] (!)`	<Rn>, <Rm>	
`LDR <Rd>, [<Rn>, <Rm>, LSL #2] (!)`	<Rn>, <Rm>	
`LDR <Rd>, [<Rn>], #cns`	<Rn>	
`LDR <Rd>, [<Rn>], <Rm>`	<Rn>, <Rm>	
`LDR <Rd>, [<Rn>], <Rm>, LSL #2`	<Rn>, <Rm>	
`<addr_md_2cycle>`		
`LDR <Rd>, [<Rn>, -<Rm>] (!)`	<Rm>	If negative register offset, or shift other than LSL #2 then two-issue cycles.
`LDR <Rd>, [Rm, -<Rm> <shf> <cns>] (!)`	<Rm>	
`LDR <Rd>, [<Rn>], -<Rm>`	<Rm>	
`LDR <Rd>, [<Rn>], -<Rm> <shf> <cns>`	<Rm>	

14.10.1 Base register update

The base register update for load or store instructions occurs in the ALU pipeline. To prevent an interlock for back-to-back load or store instructions reusing the same base register, there is a local forwarding path to recycle the updated base register around the ADD stage.

For example, the following instruction sequence take three cycles to execute:

```
LDR r5, [r2, #4]!
LDR r6, [r2, #0x10]!
LDR r7, [r2, #0x20]!
```

14.11 Load and store double instructions

This section describes the cycle timing behavior for the LDRD and STRD instructions

The LDRD and STRD instructions:

- Are two-cycle issue if either a negative register offset or a shift other than LSL #2 is used. Only the offset register is an Early Reg.
- Are single-cycle issue if either a constant offset is used or if a register offset with no shift, or shift by 2 is used. Both the base and any offset register are Early Regs.
- Take only one memory cycle if the address is doubleword aligned.
- Take two memory cycles if the address is not doubleword aligned.

The updated base register has a result latency of one. For back-to-back load/store instructions with base write back, the updated base is available to the following load/store instruction with a result latency of 0.

To prevent instructions after a STRD from writing to a register before it has stored that register, the STRD registers have a lock latency that determines how many cycles it is before a subsequent instruction which writes to that register can start.

Table 14-16 shows the cycle timing behavior for LDRD and STRD instructions.

Table 14-16 Load and store double instructions cycle timing behavior

Example instruction	Cycles	Memory cycles	Result latency (LDRD)	Register lock latency (STRD)
Address is double-word aligned				
`LDRD r1, <addr_md_1cycle>`[a]	1	1	3/3	1,2
`LDRD r1, <addr_md_2cycle>`[a]	2	2	4/4	2,3
Address not double-word aligned				
`LDRD r1, <addr_md_1cycle>`[a]	1	2	3/4	1,2
`LDRD r1, <addr_md_2cycle>`[a]	2	3	4/5	2,3

a. See Table 14-17 on page 14-24 for an explanation of `<addr_md_1cycle>` and `<addr_md_2cycle>`.

Table 14-17 on page 14-24 shows the explanation of `<addr_md_1cycle>` and `<addr_md_2cycle>` used in Table 14-16.

Table 14-17 <addr_md_1cycle> and <addr_md_2cycle> LDRD example instruction

Example instruction	Early Reg	Comment
`<addr_md_1cycle>`		
`LDRD <Rd>, [<Rn>, #cns] (!)`	<Rn>	If an immediate offset, or a positive register offset with no shift or shift LSL #2, then one-issue cycle.
`LDRD <Rd>, [<Rn>, <Rm>] (!)`	<Rn>, <Rm>	
`LDRD <Rd>, [<Rn>, <Rm>, LSL #2] (!)`	<Rn>, <Rm>	
`LDRD <Rd>, [<Rn>], #cns`	<Rn>	
`LDRD <Rd>, [<Rn>], <Rm>`	<Rn>, <Rm>	
`LDRD <Rd>, [<Rn>], <Rm>, LSL #2`	<Rn>, <Rm>	
`<addr_md_2cycle>`		
`LDRD <Rd>, [<Rn>, -<Rm>] (!)`	<Rm>	If negative register offset, or shift other than LSL #2 then two-issue cycles.
`LDRD Rd, [<Rm>, -<Rm> <shf> <cns>] (!)`	<Rm>	
`LDRD <Rd>, [<Rn>], -<Rm>`	<Rm>	
`LDRD< Rd>, [Rn], -<Rm> <shf> <cns>`	<Rm>	

14.12 Load and store multiple instructions

This section describes the cycle timing behavior for the LDM and STM instructions.

These instructions take one cycle to issue but then use multiple memory cycles to load or store all the registers. Because the memory datapath is 64-bits wide, two registers can be loaded or stored on each cycle. Following non-dependent, non-memory instructions can execute in the integer pipeline while these instructions complete. A dependent instruction is one that either:

- writes a register that has not yet been stored
- reads a register that has not yet been loaded.

Before a load or store multiple can begin all the registers in the register list must be available. For example, a STM cannot begin until all outstanding loads for registers in the register list have completed.

To prevent instructions after a store multiple from writing to a register before a store multiple has stored that register, the register list has a lock latency that determines how many cycles it is before a subsequent instruction which writes to that register can start.

14.12.1 Load and store multiples, other than load multiples including the PC

In all cases the base register, Rx, is an Early Reg.

Table 14-18 shows the cycle timing behavior of load and store multiples including the PC.

Table 14-18 Load and store multiples, other than load multiples including the PC

Example instruction	Cycles	Memory cycles	Result latency (LDM)	Register lock latency (STM)
First address 64-bit aligned				
`LDMIA Rx,{r1}`	1	1	3	1
`LDMIA Rx,{r1,r2}`	1	1	3,3	1,2
`LDMIA Rx,{r1,r2,r3}`	1	2	3,3,4	1,2,2
`LDMIA Rx,{r1,r2,r3,r4}`	1	2	3,3,4,4	1,2,2,3
`LDMIA Rx,{r1,r2,r3,r4,r5}`	1	3	3,3,4,4,5	1,2,2,3,3
`LDMIA Rx,{r1,r2,r3,r4,r5,r6}`	1	3	3,3,4,4,5,5	1,2,2,3,3,4
`LDMIA Rx,{r1,r2,r3,r4,r5,r6,r7}`	1	4	3,3,4,4,5,5,6	1,2,2,3,3,4,4

Table 14-18 Load and store multiples, other than load multiples including the PC (continued)

Example instruction	Cycles	Memory cycles	Result latency (LDM)	Register lock latency (STM)
First address not 64-bit aligned				
`LDMIA Rx,{r1}`	1	1	3	1
`LDMIA Rx,{r1,r2}`	1	2	3,4	1,2
`LDMIA Rx,{r1,r2,r3}`	1	2	3,4,4	1,2,2
`LDMIA Rx,{r1,r2,r3,r4}`	1	3	3,4,4,5	1,2,2,3
`LDMIA Rx,{r1,r2,r3,r4,r5}`	1	3	3,4,4,5,5	1,2,2,3,4
`LDMIA Rx,{r1,r2,r3,r4,r5,r6}`	1	4	3,4,4,5,5,6	1,2,2,3,4,4
`LDMIA Rx,{r1,r2,r3,r4,r5,r6,r7}`	1	4	3,4,4,5,5,6,6	1,2,2,3,4,4,5

14.12.2 Load multiples, where the PC is in the register list

If a LDM loads the PC then the PC access is performed first to accelerate the branch, followed by the rest of the register loads. The cycle timings and all register load latencies for LDMs with the PC in the list are one greater than the cycle times for the same LDM without the PC in the list.

The MP11 CPUs include a three-entry return stack which can predict procedure returns. Any LDM to the PC with the stack pointer (r13) as the base register, and which does not restore the SPSR to the CPSR, is predicted as a procedure return.

For condition code failing cycle counts, the cycles for the non-PC destination variants must be used. These are all single-cycle issue, consequently a condition code failing LDM to the PC takes one cycle.

In all cases the base register, Rx, is an Early Reg, and requires an extra cycle of result latency to provide its value.

Table 14-19 shows the cycle timing behavior of load multiples, where the PC is in the register list.

Table 14-19 Cycle timing behavior of load multiples, where the PC is in the register list

Example instruction	Cycles	Memory Cycles	Result Latency	Comments
`LDMIA sp!,{...,pc}`	4	1+n[a]	4,...	Correctly return stack predicted
`LDMIA sp!,{...,pc}`	9	1+n[a]	4,...	Return stack mispredicted
`LDMIA <cond> sp!,{...,pc}`	9	1+n[a]	4,...	Conditional return, or empty return stack
`LDMIA rx,{...,pc}`	8	1+n[a]	4,...	Not return stack predicted

a. Where n is the number of memory cycles for this instruction if the PC had not been in the register list.

14.12.3 Example interlocks

The following sequence that has an LDM instruction take five cycles, because r3 has a result latency of four cycles:

```
LDMIA r0, {r1-r7}
ADD r10, r10, r3
```

The following that has an STM instruction takes five cycles to execute, because r6 has a register lock latency of four cycles:

```
STMIA  r0, {r1-r7}
ADD    r6, r10, r11
```

14.13 RFE and SRS instructions

This section describes the cycle timing for the RFE and SRS instructions.

These instructions return from an exception and save exception return state respectively. The RFE instruction always requires two memory cycles. It first loads the SPSR value from the stack, and then the return address. The SRS instruction takes one or two memory cycles depending on double-word alignment of the first address location.

In all cases the base register is an Early Reg, and requires an extra cycle of result latency to provide its value.

Table 14-20 shows the cycle timing behavior for RFE and SRS instructions.

Table 14-20 RFE and SRS instructions cycle timing behavior

Example instruction	Cycles	Memory Cycles
Address double-word aligned		
`RFEIA <Rn>`	9	2
`SRSIA #<mode>`	1	1
Address not double-word aligned		
`RFEIA <Rn>`	9	2
`SRSIA #<mode>`	1	2

14.14 Synchronization instructions

This section describes the cycle timing behavior for the SWP, SWPB, LDREX, and STREX instructions

In all cases the base register, Rn, is an Early Reg, and requires an extra cycle of result latency to provide its value. Table 14-21 shows the synchronization instructions cycle timing behavior.

Table 14-21 Synchronization instructions cycle timing behavior

Instruction	Cycles	Memory cycles	Result latency
`SWP Rd, <Rm>, [Rn]`	2	2	3
`SWPB Rd, <Rm>, [Rn]`	2	2	3
`LDREX <Rd>, [Rn]`	1	1	3
`STREX, Rd>, <Rm>, [Rn]`	1	1	3

14.15 Coprocessor instructions

This section describes the cycle timing behavior for the CDP, LDC, STC, LDCL, STCL, MCR, MRC, MCRR, and MRRC instructions.

The precise timing of coprocessor instructions is tightly linked with the behavior of the relevant coprocessor. The numbers below are best case numbers. For LDC or STC instructions the coprocessor can determine how many words are required. Table 14-22 shows the coprocessor instructions cycle timing behavior.

Table 14-22 Coprocessor instructions cycle timing behavior

Instruction	Cycles	Memory cycles	Result latency
`MCR`	1	1	-
`MCRR`	1	1	-
`MRC`	1	1	3
`MRRC`	1	1	3/3
`LDC` or `LDCL`	1	As required	-
`STC` or `STCL`	1	As required	-
`CDP`	1	1	-

14.16 SWI, BKPT, Undefined, and Prefetch Aborted instructions

This section describes the cycle timing behavior for SWI, Undefined instruction, BKPT and Prefetch Abort.

In all cases the exception is taken in the WBex stage of the pipeline. SWI and most Undefined instructions that fail their condition codes take one cycle. A small number of Undefined instructions that fail their condition codes take two cycles. Table 14-23 shows the SWI, BKPT, Undefined, Prefetch Aborted instructions cycle timing behavior.

Table 14-23 SWI, BKPT, Undefined, Prefetch Aborted instructions cycle timing behavior

Instruction	Cycles
SWI	8
BKPT	8
Prefetch Abort	8
Undefined instruction	8

14.17 Thumb instructions

The cycle timing behavior for Thumb instructions follow the ARM equivalent instruction cycle timing behavior.

Thumb BL instructions that are encoded as two Thumb instructions, can be dynamically predicted. The predictions occurs on the second part of the BL pair, consequently a correct prediction takes two cycles.

Part B

Vector Floating-Point

Chapter 15
Introduction to VFP

This chapter introduces the VFP11 coprocessor. It contains the following sections:

15.1 About the VFP11 coprocessor

The VFP11 coprocessor is an implementation of the *ARM Vector Floating-point Architecture* (VFPv2). It provides low-cost floating-point computation that is fully compliant with the *ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, referred to in this document as the IEEE 754 standard. The VFP11 coprocessor supports all addressing modes described in section C5 of the *ARM Architecture Reference Manual*.

The VFP11 coprocessor is optimized for:

- high data transfer bandwidth through 64-bit split load and store buses
- fast hardware execution of a high percentage of operations on normalized data, resulting in higher overall performance while providing full IEEE 754 standard support when required
- hardware divide and square root operations in parallel with other arithmetic operations to reduce the impact of long-latency operations
- near IEEE 754 standard compatibility in RunFast mode without support code assistance, providing determinable run-time calculations for all input data
- low power consumption, small die size, and reduced kernel code.

The VFP11 coprocessor is an ARM enhanced numeric coprocessor that provides IEEE 754 standard-compatible operations. Designed for the ARM11 family of cores, the VFP11 coprocessor fully supports single-precision and double-precision add, subtract, multiply, divide, multiply and accumulate, and square root operations. Conversions between floating-point data formats and ARM integer word format are provided, with special operations to perform the conversion in round-towards-zero mode for high-level language support.

The VFP11 coprocessor provides a performance-power-area solution for embedded applications and high performance for general-purpose applications.

Note

This manual describes only VFP11-specific implementation issues. Refer also to the Vector Floating-point Architecture section of the *ARM Architecture Reference Manual*.

15.2 Applications

The VFP11 coprocessor provides floating-point computation suitable for a wide spectrum of applications such as:

- personal digital assistants and smartphones for graphics, voice compression and decompression, user interfaces, Java interpretation, and *Just In Time* (JIT) compilation
- games machines for three-dimensional graphics and digital audio
- printers and *MultiFunction Peripheral* (MFP) controllers for high-definition color rendering
- set-top boxes for digital audio and digital video, and three-dimensional user interfaces
- automotive applications for engine management and power train computations.

15.3 Coprocessor interface

The VFP11 coprocessor is integrated with an ARM11 processor through a dedicated VFP coprocessor interface.

The VFP11 coprocessor uses coprocessor ID number 10 for single-precision instructions and coprocessor ID number 11 for double-precision instructions. In some cases, such as mixed-precision instructions, the coprocessor ID represents the destination precision. In a system containing a VFP11 coprocessor, these coprocessor ID numbers must not be used by another coprocessor.

Access to the VFP11 coprocessor is controlled by the ARM11 Coprocessor Access Control Register. The coprocessor access rights must be configured correctly before any VFP11 instructions can be executed. For more detailed information, refer to Part A *ARM11 MPCore Processor*.

15.4 VFP11 coprocessor pipelines

The VFP11 coprocessor has three separate instruction pipelines:

- the *Multiply and Accumulate* (FMAC) pipeline
- the *Divide and Square root* (DS) pipeline
- the *Load/Store* (LS) pipeline.

Each pipeline can operate independently of the other pipelines and in parallel with them. Each of the three pipelines shares the first two pipeline stages, Decode and Issue. These two stages and the first cycle of the Execute stage of each pipeline remain in lockstep with the ARM11 pipeline stage but effectively one cycle behind the ARM11 pipeline. When the ARM11 processor is in the Issue stage for a particular VFP instruction, the VFP11 coprocessor is in the Decode stage for the same instruction. This lockstep mechanism maintains in-order issue of instructions between the ARM11 processor and the VFP11 coprocessor.

The three pipelines can operate in parallel, enabling more than one instruction to be completed per cycle. Instructions issued to the FMAC pipeline can complete out of order with respect to operations in the LS and DS pipelines. This out-of-order completion might be visible to the user when a short vector FMAC or DS operation generates an exception, and an LS operation begins before the exception is detected. The destination registers or memory of the LS operation reflect the completion of a transfer. The destination registers of the exceptional FMAC or DS operation retain the values they had before the operation started. This is described in more detail in *Parallel execution* on page 18-23.

Except for divide and square root operations, the pipelines support single-cycle throughput for all single-precision operations and most double-precision operations. Double-precision multiply and multiply and accumulate operations have a two-cycle throughput. The LS pipeline is capable of supplying two single-precision operands or one double-precision operand per cycle, balancing the data transfer capability with the operand requirements.

15.4.1 FMAC pipeline

Figure 15-1 on page 15-6 shows the structure of the FMAC pipeline.

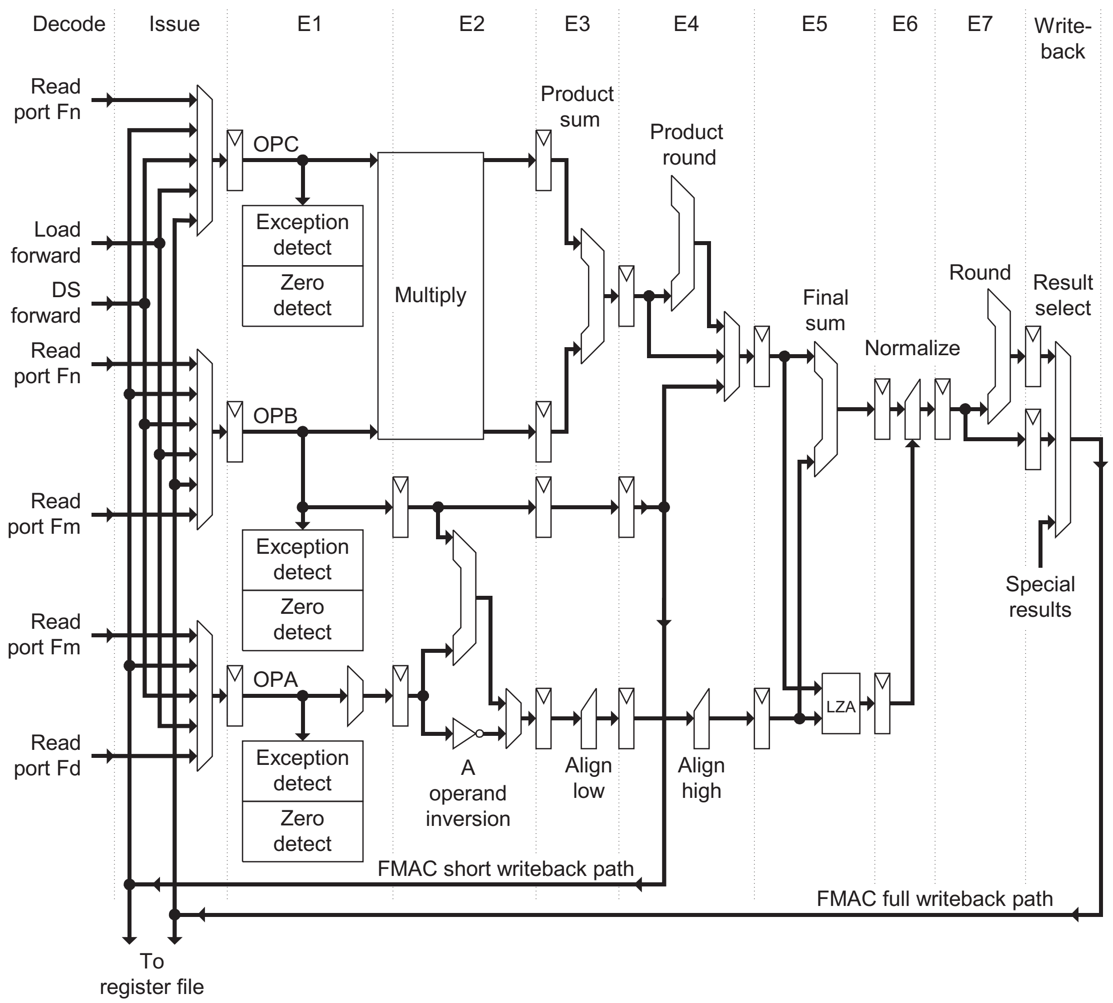


Figure 15-1 FMAC pipeline

FMAC pipeline instructions

The FMAC pipeline executes the following instructions:

FADD	Add.
FSUB	Subtract.
FMUL	Multiply.
FNMUL	Negated multiply.

FMAC	Multiply and accumulate.
FNMAC	Negated multiply and accumulate.
FMSC	Multiply and subtract.
FNMSC	Negated multiply and subtract.
FABS	Absolute value.
FNEG	Negation.
FUITO	Convert unsigned integer to float.
FTOUI	Convert float to unsigned integer.
FSITO	Convert signed integer to float.
FTOSI	Convert float to signed integer.
FTOUIZ	Convert float to unsigned integer with forced round-towards-zero mode.
FTOSIZ	Convert float to signed integer with forced round-towards-zero mode.
FCMP	Compare.
FCMPE	Compare (NaN exceptions).
FCMPZ	Compare with zero.
FCMPEZ	Compare with zero (NaN exceptions).
FCVTSD	Convert from double-precision to single-precision.
FCVTDS	Convert from single-precision to double-precision.
FCPY	Copy register.

See *Execution timing* on page 18-25 for cycle counts.

The FMAC family of instructions (FMAC, FNMAC, FMSC, and FNMSC) perform a chained multiply and accumulate operation. The product is computed, rounded according to the specified rounding mode and destination precision, and checked for exceptions before the accumulate operation is performed. The accumulate operation is also rounded according to the specified rounding mode and destination precision and checked for exceptions. The final result is identical to the equivalent sequence of operations executed in sequence. Exception processing and status reporting also reflect the independence of the components of the chained operations.

As an example, the FMAC instruction performs a chained multiply and add operation with the following sequence of operations:

1. The product of the operands in the Fn and Fm registers is computed.
2. The product is rounded according to the current rounding mode and destination precision and checked for exceptions.
3. The result is summed with the operand in the Fd register.

4. The sum is rounded according to the current rounding mode and destination precision and checked for exceptions. If no exception conditions that require support code are present, the result is written to the Fd register.

   For example, the following two operations return the same result:

```
FMACS S0, S1, S2


FMULS TEMP, S1, S2
FADDS S0, S0, TEMP
```

15.4.2 DS pipeline

Figure 15-2 shows the structure of the DS pipeline.

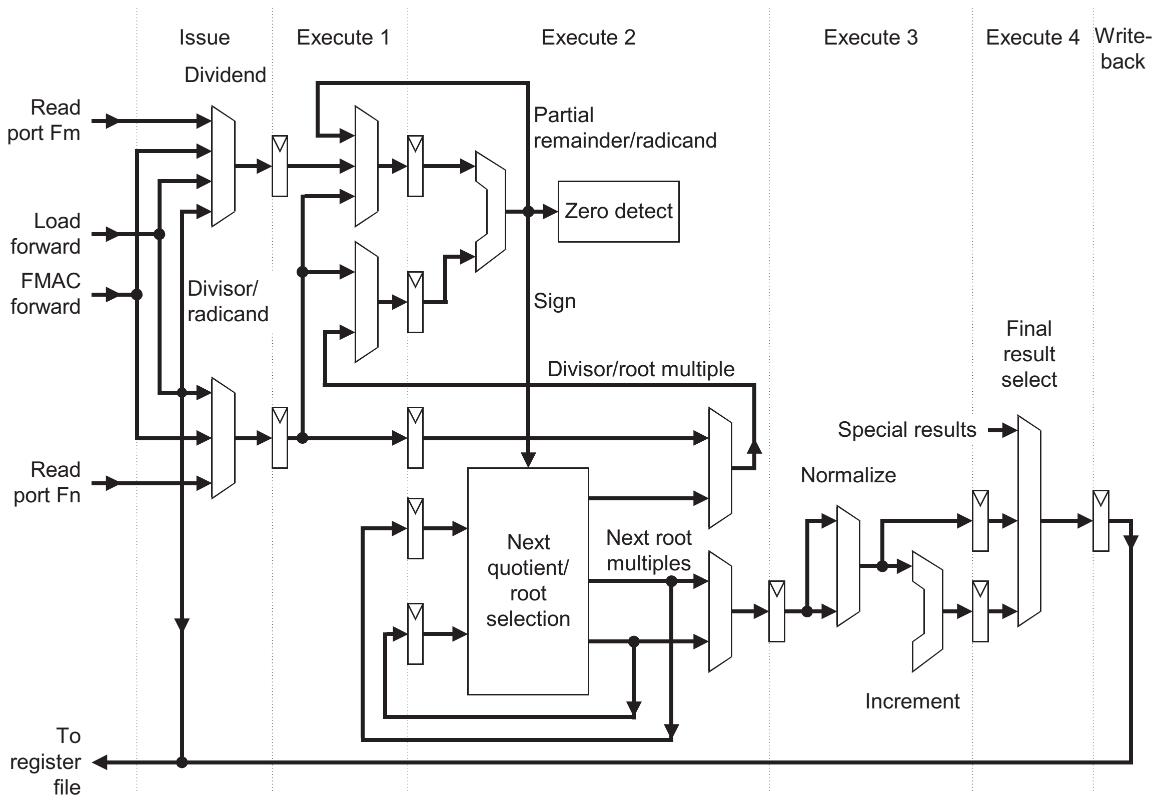


Figure 15-2 DS pipeline

DS pipeline instructions

The DS pipeline executes the following instructions:

FDIV	Divide.
FSQRT	Square root.

The VFP11 coprocessor executes divide and square root instructions for both single-precision and double-precision operands with all IEEE 754 standard rounding modes supported. The DS unit uses a shared radix-4 algorithm that provides a good balance between speed and chip area. DS operations have a latency of 19 cycles for single-precision operations and 33 cycles for double-precision operations. The throughput is 15 cycles for single-precision operations and 29 cycles for double-precision operations.

15.4.3 LS pipeline

The LS pipeline handles all of the instructions that involve data transfer to and from the ARM11 processor, including loads, stores, moves to coprocessor system registers, and moves from coprocessor system registers. It remains synchronized with the ARM11 LS pipeline for the duration of the instruction.

Data written to the ARM11 processor is read from the VFP11 coprocessor register file in the Issue stage and transferred to the ARM11 processor in the next cycle and is latched on the ARM11 data cache1/data cache 2 cycle boundary. The transfer is made on a dedicated 64-bit store data bus between the VFP11 coprocessor and the ARM11 processor.

Load data is written to the VFP11 coprocessor on a dedicated 64-bit load bus between the ARM11 processor and all coprocessors. Data is received by the VFP11 coprocessor in the Writeback stage. Data is written to the register file in the Writeback stage, and available for forwarding to data processing operations in the same cycle. Figure 15-3 on page 15-10 shows the structure of the LS pipeline.

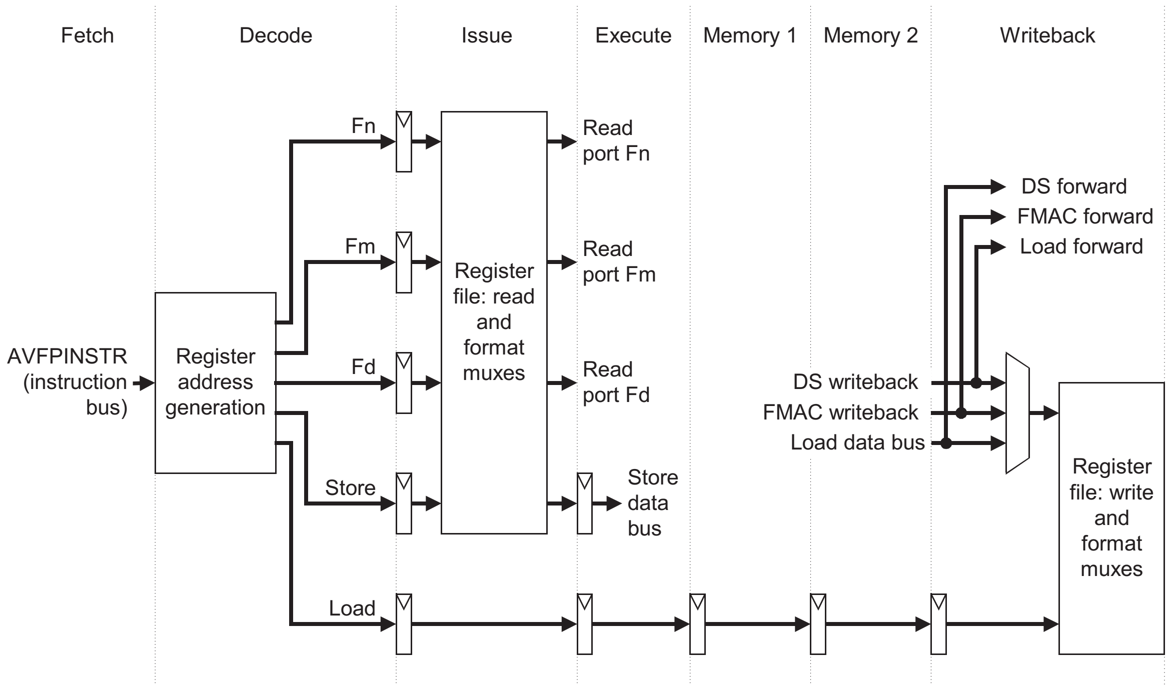


Figure 15-3 LS pipeline

LS pipeline instructions

The LS pipeline executes the following instructions:

FLD	Load a single-precision, double-precision, or 32-bit integer value from memory to the VFP11 register file.
FLDM	Load up to 32 single-precision or integer values or 16 double-precision values from memory to the VFP11 register file.
FST	Store a single-precision, double-precision, or 32-bit integer value from the VFP11 register file to memory.
FSTM	Store up to 32 single-precision or integer values or 16 double-precision values from the VFP11 register file to memory.
FMSR	Move a single-precision or integer value from an ARM11 register to a VFP11 single-precision register.

FMRS	Move a single-precision or integer value from a VFP11 single-precision register to an ARM11 register.
FMDHR	Move an ARM11 register value to the upper half of a VFP11 double-precision register.
FMDLR	Move an ARM11 register value to the lower half of a VFP11 double-precision register.
FMRDH	Move the upper half of a double-precision value from a VFP11 double-precision register to an ARM11 register.
FMRDL	Move the lower half of a double-precision value from a VFP11 double-precision register to an ARM11 register.
FMDRR	Move two ARM11 register values to a VFP11 double-precision register.
FMRRD	Move a double-precision VFP11 register value to two ARM11 registers.
FMSRR	Move two ARM11 register values to two consecutively-numbered VFP11 single-precision registers.
FMRRS	Move two consecutively-numbered VFP11 single-precision register values to two ARM11 registers.
FMXR	Move an ARM11 register value to a VFP11 control register.
FMRX	Move a VFP11 control register value to an ARM11 register.
FMSTAT	Move N, C, Z, and V flags from the VFP11 FPSCR to the ARM11 CPSR.

15.5 Modes of operation

The VFP11 coprocessor provides full IEEE 754 standard compatibility through a combination of hardware and software. There are rare cases that require significant additional compute time to resolve correctly according to the requirements of the IEEE 754 standard. For instance, the VFP11 coprocessor does not process subnormal input values directly. To provide correct handling of subnormal inputs according to the IEEE 754 standard, a trap is made to support code to process the operation. Using the support code for processing this operation can require hundreds of cycles. In some applications this is unavoidable, because compliance with the IEEE 754 standard is essential to proper operation of the program. In many other applications, strict compliance to the IEEE 754 standard is unnecessary, while determinable runtime, low interrupt latency, and low power are of more importance. To accommodate a variety of applications, the VFP11 coprocessor provides four modes of operation:

- *Full-compliance mode*
- *Flush-to-zero mode* on page 15-13
- *Default NaN mode* on page 15-13
- *RunFast mode* on page 15-13.

15.5.1 Full-compliance mode

When the VFP11 coprocessor is in full-compliance mode, all operations that cannot be processed according to the IEEE 754 standard use support code for assistance. The operations requiring support code are:

- Any CDP operation involving a subnormal input when not in flush-to-zero mode. Enable flush-to-zero mode by setting the FZ bit, FPSCR[24].
- Any CDP operation involving a NaN input when not in default NaN mode. Enable default NaN mode by setting the DN bit, FPSCR[25].
- Any CDP operation that has the potential of generating an underflow condition when not in flush-to-zero mode.
- Any CDP operation when Inexact exceptions are enabled. Enable Inexact exceptions by setting the IXE bit, FPSCR[12].
- Any CDP operation that can cause an overflow while Overflow exceptions are enabled. Enable Overflow exceptions by setting the OFE bit, FPSCR[10].
- Any CDP operation that involves an invalid arithmetic operation or an arithmetic operation on a signaling NaN when Invalid Operation exceptions are enabled. Enable Invalid Operation exceptions by setting the IOE bit, FPSCR[8].

- A float-to-integer conversion that has the potential to create an integer that cannot be represented in the destination integer format when Invalid Operation exceptions are enabled.

The support code:

- determines the nature of the exception
- determines if processing is required to perform the computation
- calls a function to compute the result and status
- transfers control to the user trap handler if the enable bit is set for a detected exception
- writes the result to the destination register, updates the FPSCR register, and returns to the user code if no enabled exception is detected
- passes control to the user trap handler and supplies any specified intermediate result for the exception if an enabled exception is detected.

Arithmetic exceptions on page 19-25 describes the conditions under which the VFP11 coprocessor traps to support code.

15.5.2 Flush-to-zero mode

Setting the FZ bit, FPSCR[24], enables flush-to-zero mode and increases performance on very small inputs and results. In flush-to-zero mode, the VFP11 coprocessor treats all subnormal input operands of arithmetic CDP operations as positive zeros in the operation. Exceptions that result from a zero operand are signaled appropriately. FABS, FNEG, FCPY, and FCMP are not considered arithmetic CDP operations and are not affected by flush-to-zero mode. A result that is *tiny*, as described in the IEEE 754 standard, for the destination precision is smaller in magnitude than the minimum normal value *before rounding* and is replaced with a positive zero. The IDC flag, FPSCR[7], indicates when an input flush occurs. The UFC flag, FPSCR[3], indicates when a result flush occurs.

15.5.3 Default NaN mode

Setting the DN bit, FPSCR[25], enables default NaN mode. In default NaN mode, the result of any operation that involves an input NaN or generated a NaN result returns the default NaN. Propagation of the fraction bits is maintained only by FABS, FNEG, and FCPY operations, all other CDP operations ignore any information in the fraction bits of an input NaN. See *NaN handling* on page 17-5 for a description of default NaNs.

15.5.4 RunFast mode

RunFast mode is the combination of the following conditions:

- the VFP11 coprocessor is in flush-to-zero mode

- the VFP11 coprocessor is in default NaN mode
- all exception enable bits are cleared.

In RunFast mode the VFP11 coprocessor:

- processes subnormal input operands as positive zeros
- processes results that are tiny before rounding, that is, between the positive and negative minimum normal values for the destination precision, as positive zeros
- processes input NaNs as default NaNs
- returns the default result specified by the IEEE 754 standard for overflow, division by zero, invalid operation, or inexact operation conditions fully in hardware and without additional latency
- processes all operations in hardware without trapping to support code.

RunFast mode enables the programmer to write code for the VFP11 coprocessor that runs in a determinable time without support code assistance, regardless of the characteristics of the input data. In RunFast mode, no user exception traps are available. However, the exception flags in the FPSCR register are compliant with the IEEE 754 standard for Inexact, Overflow, Invalid Operation, and Division by Zero exceptions. The underflow flag is modified for flush-to-zero mode. Each of these flags is set by an exceptional condition and can by cleared only by a write to the FPSCR register.

15.6 Short vector instructions

The VFPv2 architecture supports execution of *short vector* instructions of up to eight operations on single-precision data and up to four operations on double-precision data. Short vectors are most useful in graphics and signal-processing applications. They reduce code size, increase speed of execution by supporting parallel operations and multiple transfers, and simplify algorithms with high data throughput.

Short vector operations issue the individual operations specified in the instruction in a serial fashion. To eliminate data hazards, short vector operations begin execution only after all source registers are available, and all destination registers are not targets of other operations.

See Chapter 18 *VFP Instruction Execution* for more information on execution of short vector instructions.

15.7 Parallel execution of instructions

The VFP11 coprocessor provides the ability to execute several floating-point operations in parallel, while the ARM11 processor is executing ARM instructions. While a short vector operation executes for a number of cycles in the VFP11 coprocessor, it appears to the ARM11 processor as a single-cycle instruction and is retired in the ARM11 processor before it completes execution in the VFP11 coprocessor.

The three pipelines are designed to operate independently of one another once initial processing is completed. This makes it possible to issue a short vector operation and a load or store multiple operation in the next cycle and have both executing at the same time, provided no data hazards exist between the two instructions. With this mechanism, algorithms that can be double-buffered can be written to hide much of the time to transfer data to and from the VFP11 coprocessor under the arithmetic operations, resulting in a significant improvement in performance.

The separate DS pipeline enables both data transfer operations and CDPs that are not to the DS pipeline to execute in parallel with the divide. The DS block has a dedicated write port to the register file, and no special care is required when executing operations in parallel with divide or square root instructions. This is described further in *Parallel execution* on page 18-23.

15.8 VFP11 treatment of branch instructions

The VFP11 coprocessor does not directly provide branch instructions. Instead, the result of a floating-point compare instruction can be stored in the ARM11 condition code flags using the FMSTAT instruction. This enables the ARM11 branch instructions and conditional execution capabilities to be used for executing conditional floating-point code. See section C3 of the *ARM Architecture Reference Manual* for information on the use of ARM11 conditional execution to test IEEE 754 standard predicates.

In some cases, full IEEE 754 standard comparisons are not required. Simple comparisons of single-precision data, such as comparisons to zero or to a constant, can be done using an FMRS transfer and the ARM11 CMP and CMN instructions. This method is faster in many cases than using an FCMP instruction followed by an FMSTAT instruction. For more information, see *Compliance with the IEEE 754 standard* on page 17-3 and *Comparisons* on page 17-6.

15.9 Writing optimal VFP11 code

The following guidelines provide significant performance increases for VFP11 code:

- Unless there is a read-after-write hazard, program most scalar operations to immediately follow each other. Instead of a VFP11 FMAC instruction, use either a single ARM11 instruction or a VFP11 load or store instruction after the following instructions:
  - a scalar double-precision multiply
  - a multiply and accumulate
  - a short vector instruction of length greater than one iteration.
- Avoid short vector divides and square roots. The VFP11 FMAC and DS pipelines are unavailable until the final iteration of the short vector DS operation issues from the Execute 1 stage. If the short vector DS operation can be separated, other VFP11 instructions can be issued in the cycles immediately following the divide or square root. See *Parallel execution* on page 18-23.
- The best performance for data-intensive applications requires double-buffering looped short vector instructions. The register banks can be divided to provide multiple independent working areas. To take advantage of the simultaneous execution of data transfer and short vector arithmetic instructions, follow the arithmetic instructions on one bank with load or store instructions on the other bank.
- Moves to and from control registers are serializing. Avoid placing these in loops or time-critical code.
- If fully compliant IEEE 754 standard comparisons are not required, avoid using FCMPE and FCMPEZ. Using an FMRS instruction with an ARM11 CMP or CMN can be faster for simple comparisons. See *Comparisons* on page 17-6.

15.10 VFP11 revision information

This manual describes the fourth version of the VFP11 coprocessor.

Updates in the fourth version of the VFP11 coprocessor are:

- corrections for errata
- addition of *Media and VFP Feature Registers* (MVFRs)
- update to the FPSID register to reflect the fourth version.

There are no other functional differences between the VFP11 third and fourth versions.

Chapter 16
VFP Register File

This chapter describes implementation-specific features of the VFP11 coprocessor that are useful to programmers. It contains the following sections:

16.1 About the register file

The register file is organized in four banks of eight registers. Each 32-bit register can store either a single-precision floating-point number or an integer.

Any consecutive pair of registers, $[R_{even+1}]:[R_{even}]$, can store a double-precision floating-point number. Because a load and store operation does not modify the data, the VFP11 registers can also be used as secondary data storage by another application that does not use floating-point values.

The register file can be configured as four circular buffers for use by short vector instructions in applications requiring high data throughput, such as filtering and graphics transforms. For short vector instructions, register addressing is circular within each bank. Because load and store operations do not circulate, you can load or store multiple banks, up to the entire register file, with a single instruction. Short vector operations obey certain rules specifying the conditions under which the registers in the argument list specify circular buffers or single-scalar registers. The LEN and STRIDE fields in the FPSCR register specify the number of operations performed by short vector instructions and the increment scheme within the circular register banks. Further information and examples are in Section C5 of the *ARM Architecture Reference Manual*.

16.2 Register file internal formats

The VFPv2 architecture provides the option of an internal data format that is different from some or all of the external formats. In this implementation of the VFP11 coprocessor, data in the register file has the same format as data in memory. Load or store operations for single-precision, double-precision, or integer data do not modify the format as a consequence of the transfer. However, to ensure compatibility with future VFP implementations, use FLDMX/FSTMX instructions when saving context and restoring VFP11 registers. See section C5 of the *ARM Architecture Reference Manual* for more information.

It is the responsibility of the programmer to be aware of the data type in each register. The hardware does not perform any checking of the agreement between the data type in the source registers and the data type expected by the instruction. Hardware always interprets the data according to the precision implied in the instruction.

Accessing a register that has not been initialized or loaded with valid data is Unpredictable. A way to detect access to an uninitialized register is to load all registers with *Signaling NaNs* (SNaNs) in the precision of the initial access of the register and enable the Invalid Operation exception.

16.2.1 Integer data format

The VFP11 coprocessor supports signed and unsigned 32-bit integers. Signed integers are treated as two's complement values. No modification to the data is implicit in a load, store, or transfer operation on integer data. The format of integer data within the register file is identical to the format in memory or in an MPCore general-purpose register.

16.2.2 Single-precision data format

Figure 16-1 shows the single-precision bit fields.

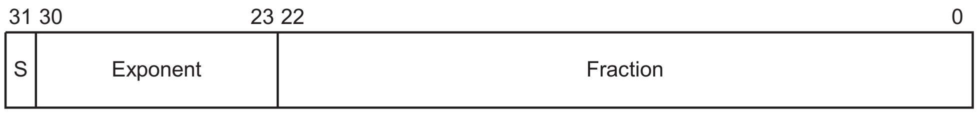


Figure 16-1 Single-precision data format

The single-precision data format contains:

- the sign bit, bit [31]
- the exponent, bits [30:23]
- the fraction, bits [22:0].

The IEEE 754 standard defines the single-precision data format of the VFP11 coprocessor. Refer to the IEEE 754 standard for details about exponent bias, special formats, and numerical ranges.

16.2.3 Double-precision data format

Double-precision format has a *Most Significant Word* (MSW) and a *Least Significant Word* (LSW). Figure 16-2 shows the double-precision bit fields.

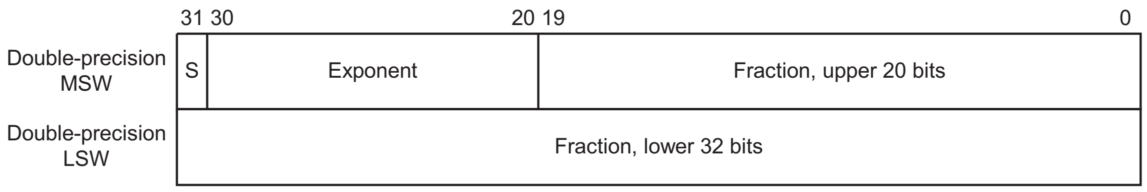


Figure 16-2 Double-precision data format

The MSW contains:

- the sign bit, bit [31]
- the exponent, bits [30:20]
- the upper 20 bits of the fraction, bits [19:0].

The LSW contains the lower 32 bits of the fraction.

The IEEE 754 standard defines the double-precision data format used in the VFP11 coprocessor. Refer to the IEEE 754 standard for details about exponent bias, special formats, and numerical ranges.

16.3 Decoding the register file

Each register file access uses the five bits of the register number in the instruction word. For single-precision and integer accesses, the most significant four bits are in the Fm, Fn, or Fd field, and the least significant bit is the M, N, or D bit for each instruction format. For instructions with double-precision operands or destinations, the M, N, and D bit corresponding to a double-precision access must be zero. Figure 16-3 shows the register file. See the *ARM Architecture Reference Manual* for instruction formats and the positions of these bits.

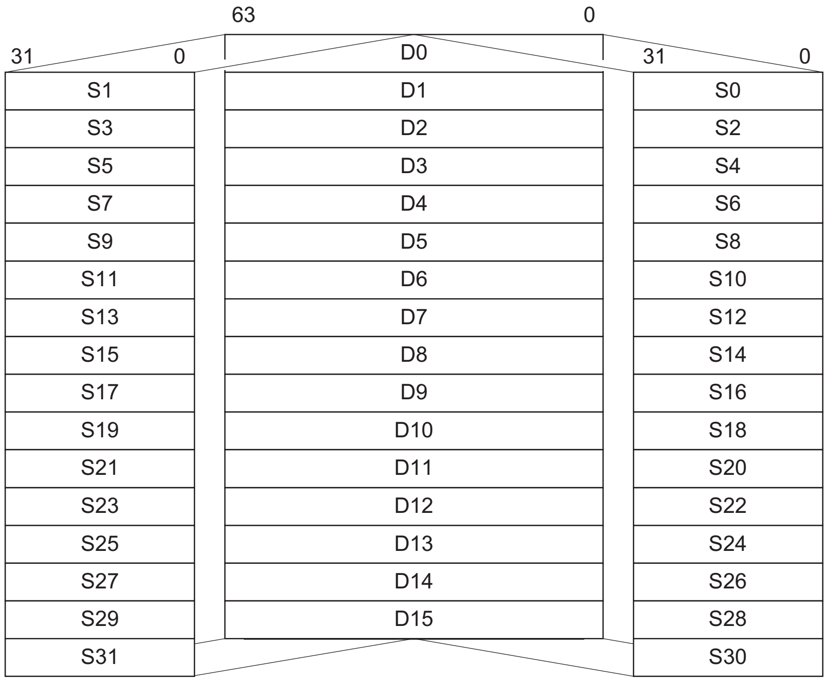


Figure 16-3 Register file access

16.4 Loading operands from MPCore registers

Floating-point data can be transferred between MPCore registers and VFP11 registers using the MCR, MRC, MCRR, and MRRC coprocessor data transfer instructions. No exceptions are possible on these transfer instructions.

MCR instructions transfer 32-bit values from MPCore registers to VFP11 registers as Table 16-1 shows.

Table 16-1 VFP11 MCR instructions

Instruction	Operation	Description
FMXR	VFP11 system register = Rd	Move from MPCore register Rd to VFP11 system register FPSID[a], FPSCR, FPEXC, FPINST, or FPINST2.
FMDLR	Dn[31:0] = Rd	Move from MPCore register Rd to lower half of VFP11 double-precision register Dn.
FMDHR	Dn[63:32] = Rd	Move from MPCore register Rd to upper half of VFP11 double-precision register Dn.
FMSR	Sn = Rd	Move from MPCore register Rd to VFP11 single-precision or integer register Sn.

a. Writing to the FPSID register does not change the contents of the FPSID but may be used as a serializing instruction.

MRC instructions transfer 32-bit values from VFP11 registers to MPCore registers as Table 16-2 shows.

Table 16-2 VFP11 MRC instructions

Instruction	Operation	Description
FMRX	Rd = VFP11 system register	Move from VFP11 system register FPSID, FPSCR, FPEXC, FPINST, or FPINST2 to MPCore register Rd.
FMRDL	Rd = Dn[31:0]	Move from lower half of VFP11 double-precision register Dn to MPCore register Rd.
FMRDH	Rd = Dn[63:32]	Move from upper half of VFP11 double-precision register Dn to MPCore register Rd.
FMRS	Rd = Sn	Move from VFP11 single-precision or integer register Sn to MPCore register Rd.

MCRR instructions transfer 64-bit quantities from MPCore registers to VFP11 registers as Table 16-3 shows.

Table 16-3 VFP11 MCRR instructions

Instruction	Operation	Description
FMDRR	Dm[31:0] = Rd Dm[63:32] = Rn	Move from MPCore registers Rd and Rn to lower and upper halves of VFP11 double-precision register Dm.
FMSRR	Sm = Rd S(m + 1) = Rn	Move from MPCore registers Rd and Rn to consecutive VFP11 single-precision registers Sm and S(m + 1).

MRRC instructions transfer 64-bit quantities from VFP11 registers to MPCore registers as Table 16-4 shows.

Table 16-4 VFP11 MRRC instructions

Instruction	Operation	Description

FMRRD	Rd = Dm[31:0]
	Rn = Dm[63:32]
		Move from lower and upper halves of VFP11 double-precision register Dm to MPCore registers Rd and Rn.

FMRRS	Rd = Sm
	Rn = S(m + 1)
		Move from single-precision VFP11 registers Sm and S(m + 1) to MPCore registers Rd and Rn.

16.5 Maintaining consistency in register precision

The VFP11 register file stores single-precision, double-precision, and integer data in the same registers. For example, D6 occupies the same registers as S12 and S13. The usable format of the register or registers depends on the last load or arithmetic instruction that wrote to the register or registers.

The VFP11 hardware does not check the register format to see if it is consistent with the precision of the current operation. Inconsistent use of the registers is possible but Unpredictable. The hardware interprets the data in the format required by the instruction regardless of the latest store or write operation to the register. It is the task of the compiler or programmer to maintain consistency in register usage.

16.6 Data transfer between memory and VFP11 registers

The B bit in the CP15 c1 Control Register (see Section B2 of the *ARM Architecture Reference Manual)*, determines whether access to stored memory is little-endian or big-endian. The MPCore processor supports both little-endian and big-endian access formats in memory.

The MPCore processor stores 32-bit words in memory with the *Least Significant Byte* (LSB) in the lowest byte of the memory address regardless of the endianness selected. For a store of a single-precision floating-point value, the LSB is located at the target address with the lower two bits of the address cleared. The *Most Significant Byte* (MSB) is at the target address with the lower two bits set. For best performance, all single-precision data must be aligned in memory to four-byte boundaries, and double-precision data must be aligned to eight-byte boundaries.

Table 16-5 shows how single-precision data is stored in memory and the address to access each byte in both little-endian and big-endian formats. In this example, the target address is `0x40000000`.

Table 16-5 Single-precision data memory images and byte addresses

Single-precision data bytes	Address in memory	Little-endian byte address	Big-endian byte address
MSB, bits [31:24]	`0x40000003`	`0x40000003`	`0x40000000`
Bits [23:16]	`0x40000002`	`0x40000002`	`0x40000001`
Bits [15:8]	`0x40000001`	`0x40000001`	`0x40000002`
LSB, bits [7:0]	`0x40000000`	`0x40000000`	`0x40000003`

For double-precision data, the location of the two words that comprise the data are stored in different locations for little-endian and big-endian data access formats. Table 16-6 shows the data storage in memory and the address to access each byte in little-endian and big-endian access modes. In this example, the target address is `0x40000000`.

Table 16-6 Double-precision data memory images and byte addresses

Double-precision data bytes	Little-endian address in memory	Little-endian byte address	Big-endian address in memory	Big-endian byte address
MSB, bits [63:56]	`0x40000007`	`0x40000007`	`0x40000003`	`0x40000000`
Bits [55:48]	`0x40000006`	`0x40000006`	`0x40000002`	`0x40000001`
Bits [47:40]	`0x40000005`	`0x40000005`	`0x40000001`	`0x40000002`
Bits [39:32]	`0x40000004`	`0x40000004`	`0x40000000`	`0x40000003`
Bits [31:24]	`0x40000003`	`0x40000003`	`0x40000007`	`0x40000004`
Bits [23:16]	`0x40000002`	`0x40000002`	`0x40000006`	`0x40000005`
Bits [15:8]	`0x40000001`	`0x40000001`	`0x40000005`	`0x40000006`
LSB, bits [7:0]	`0x40000000`	`0x40000000`	`0x40000004`	`0x40000007`

The memory image for the data is identical for both little-endian and big-endian within data words. The MPCore hardware performs the address transformations to provide both little-endian and big-endian addressing to the programmer.

16.7 Access to register banks in CDP operations

The register file is especially suited for short vector operations. The four register banks function as four circular hardware queues. Short vector operations significantly improve the performance of operations with high data throughput such as signal processing and matrix manipulation functions.

16.7.1 About register banks

As Figure 16-4 shows, the register file is divided into four banks with eight registers in each bank for single-precision instructions and four registers per bank for double-precision instructions. CDP instructions access the banks in a circular manner. Load and store multiple instructions do not access the registers in a circular manner but treat the register file as a linearly ordered structure.

See *ARM Architecture Reference Manual, Part C* for more information on VFP addressing modes.

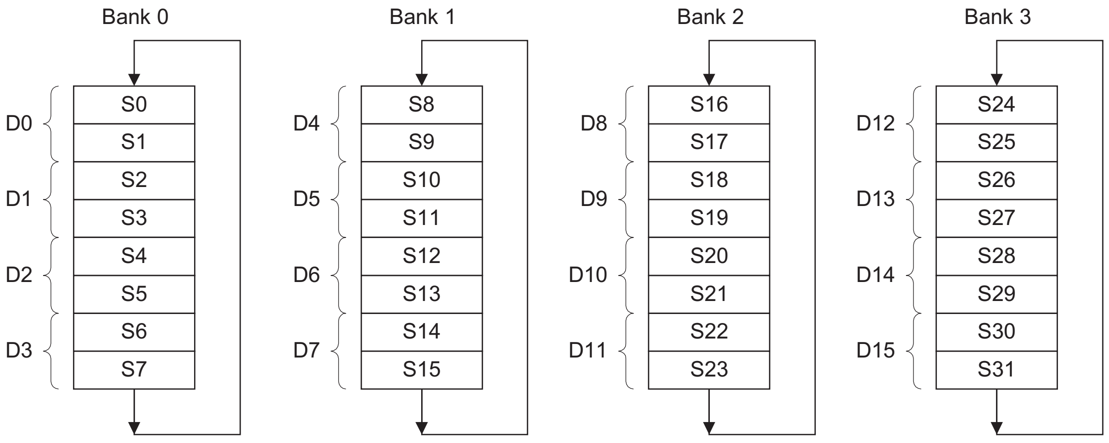


Figure 16-4 Register banks

A short vector CDP operation that has a source or destination vector crossing a bank boundary wraps around and accesses the first register in the bank.

Example 16-1 on page 16-12 shows the iterations of the following short vector add instruction:

```
FADDS S11, S22, S31
```

In this instruction, the LEN field contains b101, selecting a vector length of six iterations, and the STRIDE field contains b00, selecting a vector stride of one.

Example 16-1 Register bank wrapping

```
FADDS S11, S22, S31; 1st iteration
FADDS S12, S23, S24 ; 2nd iteration. The 2nd source vector wraps around
            ; and accesses the 1st register in the 4th bank
FADDS S13, S16, S25; 3rd iteration. The 1st source vector wraps around
            ; and accesses the 1st register in the 3rd bank
FADDS S14, S17, S26 ; 4th iteration
FADDS S15, S18, S27 ; 5th iteration
FADDS S8, S19, S28 ; 6th and last iteration. The destination vector
            ; wraps around and writes to the 1st register in the
            ; 2nd bank
```

16.7.2 Operations using register banks

The register file organization supports four types of operations described in the following sections:

- *Scalar-only instructions*
- *Short vector-only instructions* on page 16-13
- *Short vector instructions with scalar source* on page 16-13
- *Scalar instructions in short vector mode* on page 16-14.

See Chapter 17 *VFP Programmer's Model* for details of the LEN and STRIDE fields and the FPSCR register.

Scalar-only instructions

An instruction is a scalar-only operation if the operands are treated as scalars and the result is a scalar.

Clearing the LEN field in the FPSCR register selects a vector length of one iteration. For example, if the LEN field contains b000, then the following operation writes the sum of the single-precision values in S21 and S22 to S12:

```
FADDS S12, S21, S22
```

Some instructions can operate only on scalar data regardless of the value in the LEN field. These instructions are:

Compare operations

FCMPS/D, FCMPZS/D, FCMPES/D, and FCMPEZS/D.

Integer conversions

FTOUIS/D, FTOUIZS/D, FTOSIS/D, FTOSIZS/D, FUITOS/D, and FSITOS/D.

Precision conversions

FCVTDS and FCVTSD.

Short vector-only instructions

Vector-only instructions require that the value in the LEN field is nonzero, and that the destination and Fm registers are not in bank 0.

Example 16-2 shows the iterations of the following short vector instruction:

```
FMACS S16, S0, S8
```

In the example, the LEN field contains b011, selecting a vector length of four iterations, and the STRIDE field contains b00, selecting a vector stride of one.

Example 16-2 Short vector instruction

```
FMACS S16, S0, S8; 1st iteration
FMACS S17, S1, S9; 2nd iteration
FMACS S18, S2, S10; 3rd iteration
FMACS S19, S3, S11; 4th and last iteration
```

Short vector instructions with scalar source

The VFPv2 architecture enables a vector to be operated on by a scalar operand. The destination must be a vector, that is, not in bank 0, and the Fm operand must be in bank 0.

Example 16-3 shows the iterations of the following short vector instruction with a scalar source:

```
FMULD D12, D8, D2
```

In the example, the LEN field contains b001, selecting a vector length of two iterations, and the STRIDE field contains b00, selecting a vector stride of one.

Example 16-3 Short vector instruction with scalar source

```
FMULD D12, D8, D2; 1st iteration
FMULD D13, D9, D2; 2nd and last iteration
```

This scales the two source registers, D8 and D9, by the value in D2 and writes the new values to D12 and D13.

Scalar instructions in short vector mode

You can mix scalar and short vector operations by carefully selecting the source and destination registers. If the destination is in bank 0, the operation is scalar-only regardless of the value in the LEN field. You do not have to change the LEN field from a nonzero value to b000 to perform scalar operations.

Example 16-4 shows the sequence of operations for the following instructions:

```
FABSD D4, D8

FADDS S0, S0, S31

FMULS S24, S26, S1
```

In the example, the LEN field contains b001, selecting a vector length of two iterations, and the STRIDE field contains b00, selecting a vector stride of one.

Example 16-4 Scalar operation in short vector mode

```
FABSD D4, D8  ; vector DP ABS operation on regs (D8, D9) to (D4, D5)
FABSD D5, D9
FADDS S0, S0, S31  ; scalar increment of S0 by S31
FMULS S24, S26, S1 ; vector (S26, S27) scaled by S1 and written to (S24, S25)
FMULS S25, S27, S1
```

The tables that follow show the four types of operations possible in the VFPv2 architecture. In the tables, *Any* refers to the availability of all registers in the precision for the specified operand. *S* refers to a scalar operand with only a single register. *V* refers to a vector operand with multiple registers. Table 16-7 describes single-precision three-operand register usage.

Table 16-7 Single-precision three-operand register usage

LEN field	Fd	Fn	Fm	Operation type
b000	Any	Any	Any	S = S op S OR S = S ± S × S
Nonzero	0-7	Any	Any	S = S op S OR S = S ± S × S
Nonzero	8-31	Any	0-7	V = V op S OR V = V ± V × S
Nonzero	8-31	Any	8-31	V = V op V OR V = V ± V × V

Table 16-8 describes single-precision two-operand register usage.

Table 16-8 Single-precision two-operand register usage

LEN field	Fd	Fm	Operation type
b000	Any	Any	S = op S
Nonzero	0-7	Any	S = op S
Nonzero	8-31	0-7	V = op S
Nonzero	8-31	8-31	V = op V

Table 16-9 describes double-precision three-operand register usage.

Table 16-9 Double-precision three-operand register usage

LEN field	Fd	Fn	Fm	Operation type
b000	Any	Any	Any	S = S op S OR S = S ± S × S
Nonzero	0-3	Any	Any	S = S op S OR S = S ± S × S
Nonzero	4-15	Any	0-3	V = V op S OR V = V ± V × S
Nonzero	4-15	Any	4-15	V = V op V OR V = V ± V × V

Table 16-10 describes double-precision two-operand register usage.

Table 16-10 Double-precision two-operand register usage

LEN field	Fd	Fm	Operation type
b000	Any	Any	S = op S
Nonzero	0-3	Any	S = op S
Nonzero	4-15	0-3	V = op S
Nonzero	4-15	4-15	V = op V

Chapter 17
VFP Programmer's Model

This chapter describes implementation-specific features of the VFP11 coprocessor that are useful to programmers. It contains the following sections:

17.1 About the programmer's model

This section introduces the VFP11 implementation of the VFPv2 floating-point architecture.

——— **Note** ———

The *ARM Architecture Reference Manual* describes the VFPv1 architecture.

————————

The VFP11 coprocessor implements all the instructions and modes of the VFPv2 architecture. The VFPv2 architecture adds the following features and enhancements to the VFPv1 architecture:

- The ARM v5TE instruction set. This includes the MRRC and MCRR instructions to transfer 64-bit data between the MPCore processor and the VFP11 coprocessor. These instructions allow the transfer of a double-precision register or two consecutively numbered single-precision registers to or from a pair of MPCore registers. See *Loading operands from MPCore registers* on page 16-6 for syntax and usage of VFP MRRC and MCRR instructions.
- Default NaN mode. In default NaN mode, any operation involving one or more NaN operands produces the default NaN as a result, rather than returning the NaN or one of the NaNs involved in the operation. This mode is compatible with the IEEE 754 standard but not with current handling of NaNs by industry.
- Addition of the input subnormal flag, IDC (FPSCR[7]). IDC is set whenever the VFP11 coprocessor is in flush-to-zero mode and a subnormal input operand is replaced by a positive zero. It remains set until cleared by writing to the FPSCR register. A new Input Subnormal exception enable bit, IDE (FPSCR[15]), is also added. When IDE is set, the VFP11 coprocessor traps to the Undefined trap handler for an instruction that has a subnormal input operand.
- New functionality of the underflow flag, UFC (FPSCR[3]), in flush-to-zero mode. In flush-to-zero mode, UFC is set whenever a result is below the threshold for normal numbers before rounding, and the result is flushed to zero. UFC remains set until cleared by writing to the FPSCR register. Setting the Underflow exception enable bit, UFE (FPSCR[11]), does not cause a trap in flush-to-zero mode.
- New functionality of the inexact flag, IXC (FPSCR[4]), in flush-to-zero mode. In VFPv1, IXC is set when an input or result is flushed to zero. In VFPv2 architecture, the IDC and UFC flags provide this information. See *Inexact exception* on page 19-23 for more information.
- Addition of RunFast mode. See *RunFast mode* on page 15-13 for details of RunFast mode operation.

17.2 Compliance with the IEEE 754 standard

This section introduces issues related to compliance with the IEEE 754 standard:

- hardware and software components
- software-based components and their availability.

Also see Section C1 of the *ARM Architecture Reference Manual* for information about VFP architecture compliance with the IEEE 754 standard.

17.2.1 An IEEE 754 standard-compliant implementation

The VFP11 hardware and support code together provide VFPv2 floating-point instruction implementations that are compliant with the IEEE 754 standard. Unless an enabled floating-point exception occurs, it appears to the program that the floating-point instruction was executed by the hardware. If an exceptional condition occurs that requires software support during instruction execution, the instruction takes significantly more cycles than normal to produce the result. This is a common practice in the industry, and the incidence of such instructions is typically very low.

17.2.2 Complete implementation of the IEEE 754 standard

The following operations from the IEEE 754 standard are not supplied by the VFP11 instruction set:

- remainder
- round floating-point number to integer-valued floating-point number
- binary-to-decimal conversions
- decimal-to-binary conversions
- direct comparison of single-precision and double-precision values.

For complete implementation of the IEEE 754 standard, the VFP11 coprocessor and support code must be augmented with library functions that implement these operations. See *Application Note 98, VFP Support Code* for details of support code and the available library functions.

17.2.3 IEEE 754 standard implementation choices

Some of the implementation choices allowed by the IEEE 754 standard and used in the VFPv2 architecture are described in Part C of the *ARM Architecture Reference Manual*.

Further implementation choices are made within the VFP11 coprocessor about which cases are handled by the VFP11 hardware and which cases bounce to the support code.

To execute frequently encountered operations as fast as possible and minimize silicon area, handling of rarely occurring values and some exceptions is relegated to the support code. The VFP11 coprocessor supports two modes for handling rarely occurring values:

Full-compliance mode

Full-compliance mode with support code assistance is fully compliant with the IEEE 754 standard. Full-compliance mode requires the floating-point support code to handle certain operands and exceptional conditions not supported in the hardware. Although the support code gives full compliance with the IEEE 754 standard, it does increase the runtime of an application and the size of kernel code.

RunFast mode

In RunFast mode, default handling of subnormal inputs, underflows, and NaN inputs is not fully compliant with the IEEE 754 standard. No user trap handlers are allowed in RunFast mode.

When flush-to-zero and default NaN modes are enabled, and all exceptions are disabled, the VFP11 coprocessor operates in RunFast mode. While the potential loss of accuracy for very small values is present, RunFast mode removes a significant number of performance-limiting stall conditions. By not requiring the floating-point support code, RunFast mode enables increased performance of typical and optimized code and a reduction in the size of kernel code. See *Hazards* on page 18-7 for more information on performance improvements in RunFast mode.

Supported formats

The supported formats are:

- Single-precision and double-precision. No extended format is supported.
- Integer formats:
  - unsigned 32-bit integers
  - two's complement signed 32-bit integers.

NaN handling

Any single-precision or double-precision values with the maximum exponent field value and a nonzero fraction field are valid NaNs. A most significant fraction bit of zero indicates a *Signaling NaN* (SNaN). A one indicates a *Quiet NaN* (QNaN). Two NaN values are treated as different NaNs if they differ in any bit. Table 17-1 shows the default NaN values in both single and double precision.

Table 17-1 Default NaN values

	Single-precision	**Double-precision**
Sign	0	0
Exponent	`0xFF`	`0x7FF`
Fraction	bit [22] = 1 bits [21:0] are all zeros	bit [51] = 1 bits [50:0] are all zeros

Any SNaN passed as input to an operation causes an Invalid Operation exception and sets the IOC flag, FPSCR[0]. If the IOE bit, FPSCR[8], is set, control passes to a user trap handler if present. If IOE is not set, a default QNaN is written to the destination register. The rules for cases involving multiple NaN operands are in the *ARM Architecture Reference Manual*.

Processing of input NaNs for ARM floating-point coprocessors and libraries is defined as follows:

- In full-compliance mode, NaNs are handled according to the *ARM Architecture Reference Manual*. The hardware does not process the NaNs directly for arithmetic CDP instructions, but traps to the support code for all NaN processing. For data transfer operations, NaNs are transferred without raising the Invalid Operation exception or trapping to support code. For the nonarithmetic CDP instructions, FABS, FNEG, and FCPY, NaNs are copied, with a change of sign if specified in the instructions, without causing the Invalid Operation exception or trapping to support code.
- In default NaN mode, NaNs are handled completely within the hardware without support code assistance. SNaNs in an arithmetic CDP operation set the IOC flag, FPSCR[0]. NaN handling by data transfer and nonarithmetic CDP instructions is the same as in full-compliance mode. Arithmetic CDP instructions involving NaN operands return the default NaN regardless of the fractions of any NaN operands.

Table 17-2 summarizes the effects of NaN operands on instruction execution.

Table 17-2 QNaN and SNaN handling

Instruction type	Default NaN mode	With QNaN operand	With SNaN operand
Arithmetic CDP	Off	INV[a] set. Bounce to support code to process operation.	INV set. Bounce to support code to process operation.
	On	No bounce. Default NaN returns.	IOC[b] set. If IOE[c] set, bounce to Invalid Operation user trap handler. If IOE clear, default NaN returns.
Nonarithmetic CDP	Off	NaN passes to destination with sign changed as appropriate.	
	On		
FCMP(Z)	Off	INV set. Bounce to support code to process operation.	INV set. Bounce to support code to process operation.
	On	No bounce. Unordered compare.	IOC set. If IOE set, bounce to Invalid Operation user trap handler. If IOE clear, unordered compare.
FCMPE(Z)	Off	INV set. Bounce to support code to process operation.	INV set. Bounce to support code to process operation.
	On	IOC set. If IOE set, bounce to Invalid Operation user trap handler. If IOE clear, unordered compare.	IOC set. If IOE set, bounce to Invalid Operation user trap handler. If IOE clear, unordered compare.
Load/store	Off	All NaNs transferred. No bounce.	
	On		

a. INV is the Input exception flag, FPEXC[7].
b. IOC is the Invalid Operation exception flag, FPSCR[0].
c. IOE is the Invalid Operation exception enable bit, FPSCR[8].

Comparisons

Comparison results modify condition code flags in the FPSCR register. The FMSTAT instruction transfers the current condition code flags in the FPSCR register to the MPCore CPSR register. Refer to the *ARM Architecture Reference Manual* for mapping of IEEE 754 standard predicates to ARM conditions. The condition code flags used are chosen so that subsequent conditional execution of ARM instructions can test the predicates defined in the IEEE 754 standard.

The VFP11 coprocessor handles most comparisons of numeric values in hardware, generating the appropriate condition code depending on whether the result is less than, equal to, or greater than. When the VFP11 coprocessor is not in flush-to-zero mode, comparisons involving subnormal operands bounce to support code.

The VFP11 coprocessor supports:

Compare operations

The compare operations are FCMPS, FCMPZS, FCMPD, and FCMPZD.

In default NaN mode, a compare instruction involving a QNaN produces an unordered result. An SNaN produces an unordered result and generates an Invalid Operation exception. If the IOE bit, FPSCR[8], is set, the Invalid Operation user trap handler is called. When the VFP11 coprocessor is not in default NaN mode, comparisons involving NaNs bounce to support code.

Compare with exception operations

The compare with exception operations are FCMPES, FCMPEZS, FCMPED, and FCMPEZD.

In default NaN mode, a compare with exception operation involving either an SNaN or a QNaN produces an unordered result and generates an Invalid Operation exception. When the VFP11 coprocessor is not in default NaN mode, comparisons involving NaNs bounce to support code.

Some simple comparisons on single-precision data can be computed directly by the MPCore processor. If only equality or comparison to zero is needed, and NaNs are not an issue, performing the comparison in MPCore registers using CMP or CMN instructions can be faster.

If branching on the state of the Z flag is needed, you can use the following instructions for positive values:

```
FMRS Rx,Sn
CMP Rx,#0
BEQ label
```

If the input values can include negative numbers, including negative zero, you can use the following code:

```
FMRS Rx, Sn
CMP Rx, #0x80000000
CMPNE Rx, #0
BEQ label
```

Using a temporary register is even faster:

```
FMRS Rx,Sn
MOVS Rt,Rx,LSL #1
BEQ label
```

Comparisons with particular values are also possible. For example, to check if a positive value is greater or equal to +1.0, use:

```
FMRS Rx,Sn
CMP Rx,#0x3F800000
BGE label
```

When comparisons are required for double-precision values, or when IEEE 754 standard comparisons are required, it is safer to use the FCMP and FCMPE instructions with FMSTAT.

Underflow

In the generation of Underflow exceptions, the *after rounding* form of *tininess* and the *subnormalization loss* form of *loss of accuracy* as described in the IEEE 754 standard are used.

In flush-to-zero mode, results that are tiny before rounding, as described in the IEEE 754 standard, are flushed to a positive zero, and the UFC flag, FPSCR[3], is set. Support code is not involved. See Part C of the *ARM Architecture Reference Manual* for information on flush-to-zero mode.

When the VFP11 coprocessor is not in flush-to-zero mode, any operation with a risk of producing a tiny result bounces to support code. If the operation does not produce a tiny result, it returns the computed result, and the UFC flag, FPSCR[3], is not set. The IXC flag, FPSCR[4], is set if the operation is inexact. If the operation produces a tiny result, the result is a subnormal or zero value, and the UFC flag, FPSCR[3], is set. See *Underflow exception* on page 19-21 for more information on underflow handling.

Exceptions

Exceptions are taken in the VFP11 coprocessor in an imprecise manner. When exception processing begins, the states of the MPCore processor and the VFP11 coprocessor might not be the same as when the exception occurred. Exceptional instructions cause the VFP11 coprocessor to enter the exceptional state, and the next VFP11 instruction triggers exception processing. After the issue of the exceptional instruction and before exception processing begins, non-VFP11 instructions and some VFP11 instructions can be executed and retired. Any source registers involved in the exceptional instruction are preserved, and the destination register is not overwritten on entry to the support code. If the detected exception enable bit is not set, the support code returns to the program flow at the point of the trigger instruction after processing the exception. If the detected exception enable bit is set, and a user trap handler is installed, the support code passes control to the user trap handler. If the exception is overflow or underflow, the intermediate result specified by the IEEE 754 standard is made available to the user trap handler.

17.3 ARMv5TE coprocessor extensions

This section describes the syntax and usage of the four ARMv5TE architecture coprocessor extension instructions:

- *FMDRR*
- *FMRRD* on page 17-11
- *FMSRR* on page 17-12
- *FMRRS* on page 17-13.

——— **Note** ———

These instructions are implementations of the MCRR and MRRC instructions, described in Section A10 of the *ARM Architecture Reference Manual*.

———————————

17.3.1 FMDRR

FMDRR transfers data from two MPCore registers to a VFP11 double-precision register. The MPCore registers do not have to be contiguous. Figure 17-1 shows the format of the FMDRR instruction.

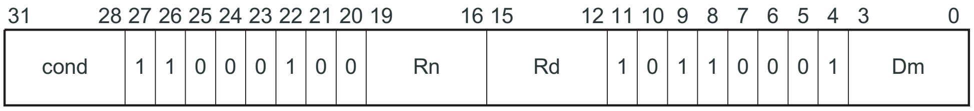


Figure 17-1 FMDRR instruction format

Syntax

```
FMDRR {<cond>} <Dm>, <Rd>, <Rn>
```

where:

`<cond>`	Is the condition under which the instruction is executed. If <cond> is omitted, the AL (always) condition is used.
`<Dm>`	Specifies the destination double-precision VFP11 coprocessor register.
`<Rd>`	Specifies the source MPCore register for the lower 32 bits of the operand.
`<Rn>`	Specifies the source MPCore register for the upper 32 bits of the operand.

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Dm[upper half] = Rn
    Dm[lower half] = Rd
```

Notes

Conversions In the programmer's model, FMDRR does not perform any conversion of the value transferred. Arithmetic instructions using either Rd or Rn treat the value as an integer, whereas most VFP instructions treat the Dm value as a double-precision floating-point number.

17.3.2 FMRRD

FMRRD transfers data in a VFP11 double-precision register to two MPCore registers. The MPCore registers do not have to be contiguous. Figure 17-2 shows the format of the FMRRD instruction.

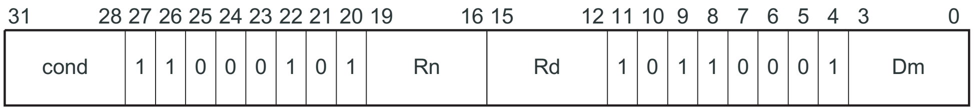


Figure 17-2 FMRRD instruction format

Syntax

```
FMRRD {<cond>} <Rd>, <Rn>, <Dm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. If <cond> is omitted, the AL (always) condition is used.
`<Rd>`	Specifies the destination MPCore register for the lower 32 bits of the operand.
`<Rn>`	Specifies the destination MPCore register for the upper 32 bits of the operand.
`<Dm>`	Specifies the source double-precision VFP11 coprocessor register.

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rn = Dm[upper half]
    Rd = Dm[lower half]
```

Notes

Use of r15 If r15 is specified for `<Rd>` or `<Rn>`, the results are Unpredictable.

Conversions In the programmer's model, FMRRD does not perform any conversion of the value transferred. Arithmetic instructions using Rd and Rn treat the contents as an integer, whereas most VFP instructions treat the Dm value as a double-precision floating-point number.

17.3.3 FMSRR

FMSRR transfers data in two MPCore registers to two consecutively numbered single-precision VFP11 registers, Sm and S(m + 1). The MPCore registers do not have to be contiguous. Figure 17-3 shows the format of the FMSRR instruction.

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	1	1	0	0	0	1	0	0	Rn	Rd	1	0	1	0	0	0	M	1	Sm

Figure 17-3 FMSRR instruction format

Syntax

```
FMSRR {<cond>} <registers>, <Rd>, <Rn>
```

where:

`<cond>` Is the condition under which the instruction is executed. If <cond> is omitted, the AL (always) condition is used.

`<registers>` Specifies the pair of consecutively numbered single-precision destination VFP11 registers, separated by a comma and surrounded by brackets. If m is the number of the first register in the list, the list is encoded in the instruction by setting Sm to the top four bits of m and M to the bottom bit of m. For example, if `<registers>` is {S1, S2}, the Sm field of the instruction is b0000 and the M bit is 1.

`<Rd>` Specifies the source MPCore register for the Sm VFP11 single-precision register.

`<Rn>` Specifies the source MPCore register for the S(m + 1) VFP11 single-precision register.

Architecture version

All

Exceptions

None

Operation

```
If ConditionPassed(cond) then
    Sm = Rd
    S(m + 1) = Rn
```

Notes

Conversions In the programmer's model, FMSRR does not perform any conversion of the value transferred. Arithmetic instructions using Rd and Rn treat the contents as an integer, whereas most VFP instructions treat the Sm and S(m + 1) values as single-precision floating-point numbers.

Invalid register lists

If Sm is b1111 and M is 1 (an encoding of S31) the instruction is Unpredictable.

17.3.4 FMRRS

FMRRS transfers data in two consecutively numbered single-precision VFP11 registers to two MPCore registers. The MPCore registers do not have to be contiguous. Figure 17-4 on page 17-14 shows the format of the FMRRS instruction.

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	1	1	0	0	0	1	0	1	Rn	Rd	1	0	1	0	0	0	M	1	Sm

Figure 17-4 FMRRS instruction format

Syntax

```
FMRRS {<cond>} <Rd>, <Rn>, <registers>
```

where:

`<cond>`	Is the condition under which the instruction is executed. If <cond> is omitted, the AL (always) condition is used.
`<Rd>`	Specifies the destination MPCore register for the Sm VFP11 coprocessor single-precision value.
`<Rn>`	Specifies the destination MPCore register for the S(m + 1) VFP11 coprocessor single-precision value.
`<registers>`	Specifies the pair of consecutively numbered single-precision VFP11 source registers, separated by a comma and surrounded by brackets. If m is the number of the first register in the list, the list is encoded in the instruction by setting Sm to the top four bits of m and M to the bottom bit of m. For example, if `<registers>` is {S16, S17}, the Sm field of the instruction is b1000 and the M bit is 0.

Architecture version

All

Exceptions

None

Operation

```
If ConditionPassed(cond) then
    Rd = Sm
    Rn = S(m + 1)
```

Notes

Conversions In the programmer's model, FMRRS does not perform any conversion of the value transferred. Arithmetic instructions using Rd and Rn treat the contents as an integer, whereas most VFP11 instructions treat the Sm and S(m + 1) values as single-precision floating-point numbers.

Invalid register lists

If Sm is b1111 and M is 1 (an encoding of S31) the instruction is Unpredictable.

Use of r15 If r15 is specified for Rd or Rn, the results are Unpredictable.

17.4 VFP11 system registers

The VFPv2 architecture describes the following three system registers that must be present in a VFP system:

- *Floating-Point System ID Register*, FPSID
- *Floating-Point Status and Control Register*, FPSCR
- *Floating-Point Exception Register*, FPEXC.

The VFP11 coprocessor provides sufficient information for processing all exceptional conditions encountered by the hardware. In an exceptional situation, the hardware provides:

- the exceptional instruction
- the instruction that might have been issued to the VFP11 coprocessor before detection of the exception
- exception status information:
  - — type of exception
  - — number of remaining short vector iterations after an exceptional iteration.

To support exceptional conditions, the VFP11 coprocessor provides two additional registers:

- *Floating-Point Instruction Register*, FPINST
- *Floating-Point Instruction Register 2*, FPINST2.

Also, the FPEXC register contains additional bits to support exceptional conditions.

These registers are designed to be used with the support code software available from ARM Limited. As a result, this document does not fully specify exception handling in all cases.

The coprocessor also provides two feature registers:

- *Media and VFP Feature Register 0* on page 17-25, MVFR0
- *Media and VFP Feature Register 1* on page 17-27, MVFR1.

Table 17-3 shows the VFP11 system registers.

Table 17-3 VFP11 system registers

Register	Access mode	Access type	Reset state
Floating-Point System ID Register, FPSID	Any	RO	`0x410120B4`
Floating-Point Status and Control Register, FPSCR	Any	R/W	`0x00000000`
Floating-Point Exception Register, FPEXC	Privileged	R/W	`0x00000000`

Table 17-3 VFP11 system registers (continued)

Register	Access mode	Access type	Reset state
Floating-Point Instruction Register, FPINST	Privileged	R/W	`0xEE000A00`
Floating-Point Instruction Register 2, FPINST2	Privileged	R/W	UNP
Media and VFP Feature Register 0, MVFR0	Any	RO	`0x11111111`
Media and VFP Feature Register 1, MVFR1	Any	RO	`0x00000000`

Use the FMRX instruction to transfer the contents of VFP11 registers to MPCore registers and the FMXR instruction to transfer the contents of MPCore registers to VFP11 registers.

Table 17-4 shows the MPCore processor modes for accessing the VFP11 system registers.

Table 17-4 Accessing VFP11 system registers

		MPCore processor mode	
Register	FMXR/FMRX <reg> field	VFP11 coprocessor enabled	VFP11 coprocessor disabled
FPSID	b0000	Any mode	Privileged mode
FPSCR	b0001	Any mode	None[a]
FPEXC	b1000	Privileged mode	Privileged mode
FPINST	b1001	Privileged mode	Privileged mode
FPINST2	b1010	Privileged mode	Privileged mode
MVFR0	b0111	Any mode	Privileged mode
MVFR1	b0110	Any mode	Privileged mode

a. An instruction that tries to access FPSCR while the VFP11 coprocessor is disabled takes the Undefined Instruction trap.

Table 17-4 shows that a privileged MPCore mode is sometimes required to access a VFP11 system register. When a privileged mode is required, an instruction that tries to access a register in a nonprivileged mode takes the Undefined Instruction trap.

The following sections describe the VFP11 system registers:

- *Floating-Point System ID Register, FPSID*
- *Floating-Point Status and Control Register, FPSCR* on page 17-19
- *Floating-point exception register, FPEXC* on page 17-22
- *Instruction registers, FPINST and FPINST2* on page 17-25.

17.4.1 Floating-Point System ID Register, FPSID

FPSID is a read-only register that identifies the VFP11 coprocessor. Figure 17-5 shows the FPSID bit fields.

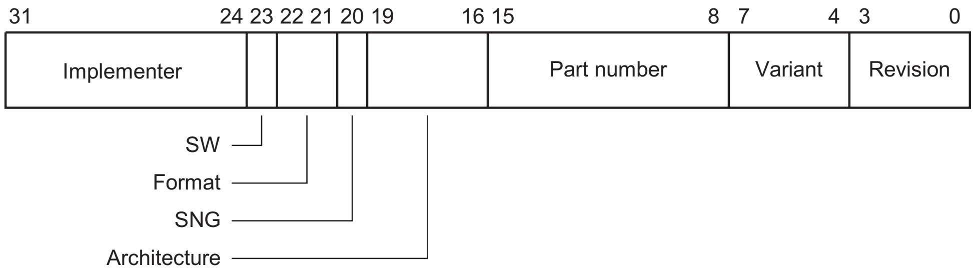


Figure 17-5 Floating-Point System ID Register

Table 17-5 describes the FPSID bit fields.

Table 17-5 FPSID bit fields

Bit	Meaning	Value
[31:24]	Implementer	`0x41` A (ARM Limited)
[23]	Hardware/software	0 Hardware implementation
[22:21]	FSTMX/FLDMX format	b00 Format 1
[20]	Precisions supported	0 Both single-precision and double-precision data supported
[19:16]	Architecture version	b0001 VFPv2 architecture

Table 17-5 FPSID bit fields (continued)

Bit	Meaning	Value
[15:8]	Part number	`0x20` VFP11
[7:4]	Variant	`0xB` MPCore VFP interface
[3:0]	Revision	`0x4` Fourth version

17.4.2 Floating-Point Status and Control Register, FPSCR

FPSCR is a read/write register that can be accessed in both privileged and unprivileged modes. All bits described as SBZ in Figure 17-6 are reserved for future expansion. They must be initialized to zeros. To ensure that these bits are not modified, code other than initialization code must use read/modify/write techniques when writing to FPSCR. Failure to observe this rule can cause Unpredictable results in future systems. Figure 17-6 shows the FPSCR bit fields.

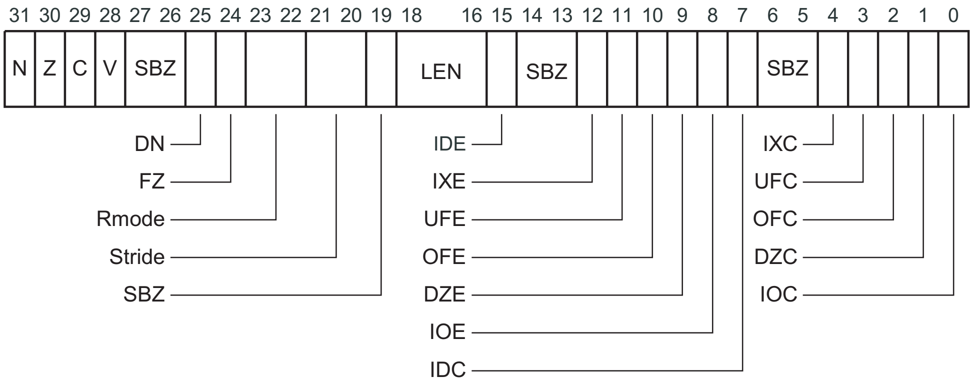


Figure 17-6 Floating-Point Status and Control Register

Table 17-6 describes the FPSCR bit fields.

Table 17-6 Encoding of the Floating-Point Status and Control Register

Bit	Name	Meaning
[31]	N	Set if comparison produces a *less than* result
[30]	Z	Set if comparison produces an *equal* result
[29]	C	Set if comparison produces an *equal*, *greater than*, or *unordered* result
[28]	V	Set if comparison produces an *unordered* result
[27:26]	-	Should Be Zero
[25]	DN	Default NaN mode enable bit: 1 = default NaN mode enabled 0 = default NaN mode disabled
[24]	FZ	Flush-to-zero mode enable bit: 1 = flush-to-zero mode enabled 0 = flush-to-zero mode disabled
[23:22]	Rmode	Rounding mode control field: b00 = Round to nearest (RN) mode b01 = Round towards plus infinity (RP) mode b10 = Round towards minus infinity (RM) mode b11 = Round towards zero (RZ) mode
[21:20]	Stride	See *Vector length and stride control* on page 17-21
[19]	-	Should Be Zero
[18:16]	LEN	See *Vector length and stride control* on page 17-21
[15]	IDE	Input Subnormal exception enable bit
[14:13]	-	Should Be Zero
[12]	IXE	Inexact exception enable bit
[11]	UFE	Underflow exception enable bit
[10]	OFE	Overflow exception enable bit
[9]	DZE	Division by Zero exception enable bit
[8]	IOE	Invalid Operation exception enable bit

Table 17-6 Encoding of the Floating-Point Status and Control Register (continued)

Bit	Name	Meaning
[7]	IDC	Input Subnormal cumulative flag
[6:5]	-	Should Be Zero
[4]	IXC	Inexact cumulative flag
[3]	UFC	Underflow cumulative flag
[2]	OFC	Overflow cumulative flag
[1]	DZC	Division by Zero cumulative flag
[0]	IOC	Invalid Operation cumulative flag

Vector length and stride control

FPSCR[18:16] is the LEN field and controls the vector length for VFP instructions that operate on short vectors. The vector length is the number of iterations in a short vector instruction.

FPSCR[21:20] is the STRIDE field and controls the vector stride. The vector stride is the increment value used to select the registers involved in the next iteration of the short vector instruction.

The rules for vector operation do not allow a vector to use the same register more than once. LEN and STRIDE combinations that use a register more than once produce Unpredictable results, as Table 17-7 shows. Some combinations that work normally in single-precision short vector instructions cause Unpredictable results in double-precision instructions.

Table 17-7 Vector length and stride combinations

LEN	Vector length	STRIDE	Vector stride	Single-precision vector instructions	Double-precision vector instructions
b000	1	b00	-	All instructions are scalar	All instructions are scalar
b000	1	b11	-	Unpredictable	Unpredictable
b001	2	b00	1	Work normally	Work normally
b001	2	b11	2	Work normally	Work normally
b010	3	b00	1	Work normally	Work normally
b010	3	b11	2	Work normally	Unpredictable
b011	4	b00	1	Work normally	Work normally
b011	4	b11	2	Work normally	Unpredictable
b100	5	b00	1	Work normally	Unpredictable
b100	5	b11	2	Unpredictable	Unpredictable
b101	6	b00	1	Work normally	Unpredictable
b101	6	b11	2	Unpredictable	Unpredictable
b110	7	b00	1	Work normally	Unpredictable
b110	7	b11	2	Unpredictable	Unpredictable
b111	8	b00	1	Work normally	Unpredictable
b111	8	b11	2	Unpredictable	Unpredictable

17.4.3 Floating-point exception register, FPEXC

In a bounce situation, the FPEXC register records the exceptional status. The FPEXC register information assists the support code in processing the exceptional condition or reporting the condition to a system trap handler or a user trap handler.

You must save and restore the FPEXC register whenever changing the context. If the EX flag, FPEXC[31], is set, then the VFP11 coprocessor is in the exceptional state, and you must also save and restore the FPINST and FPINST2 registers. You can write the context switch code to determine from the EX flag which registers to save and restore or to simply save all three.

The EN bit, FPEXC[30], is the VFP enable bit. Clearing EN disables the VFP11 coprocessor. The VFP11 coprocessor clears the EN bit on reset.

The INV flag, FPEXC[7], signals Input exceptions. An Input exception is a condition in which the hardware cannot process one or more input operands according to the architectural specifications. This includes subnormal inputs when the VFP11 coprocessor is not in flush-to-zero mode and NaNs when the VFP11 coprocessor is not in default NaN mode.

The UFC flag, FPEXC[3], is set whenever an operation has the potential to generate a result that is below the minimum threshold for the destination precision.

The OFC flag, FPEXC[2], is set whenever an operation has the potential to generate a result that, after rounding, exceeds the largest representable number in the destination format.

The IOC flag, FPEXC[0], is set whenever an operation has the potential to generate a result that cannot be represented or is not defined.

Note

To prevent an infinite loop of exceptions, the support code must clear the EX flag, FPEXC[31], immediately on entry to the exception code. All exception flags must be cleared before returning from exception code to user code.

Figure 17-7 shows the FPEXC bit fields.

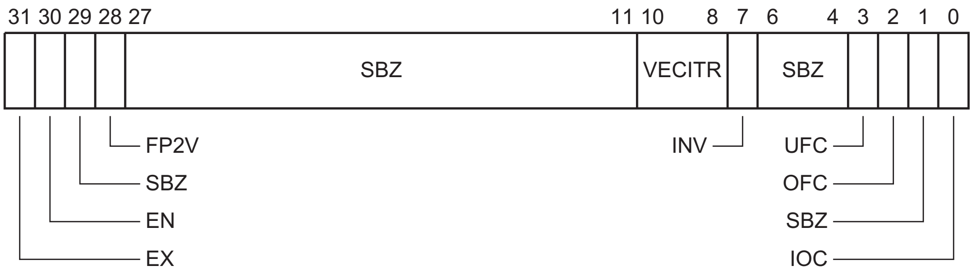


Figure 17-7 Floating-Point Exception Register

Table 17-8 describes the bit fields of the FPEXC register.

Table 17-8 Encoding of the Floating-Point Exception Register

Bit	Name	Description
[31]	EX	Exception flag. When EX is set, the VFP11 coprocessor is in the exceptional state. EX must be cleared by the exception handling routine.
[30]	EN	VFP enable bit. Setting EN enables the VFP11 coprocessor. Reset clears EN.
[29]	-	Should Be Zero.
[28]	FP2V	FPINST2 instruction valid flag. Set when FPINST2 contains a valid instruction. FP2V must be cleared by the exception handling routine.
[27:11]	-	Should Be Zero.

[10:8]	VECITR	Vector iteration count field.
		VECITR contains the number of remaining short vector iterations after a potential exception was detected in one of the iterations:
		b000 = 1 iteration
		b001 = 2 iterations
		b010 = 3 iterations
		b011 = 4 iterations
		b100 = 5 iterations
		b101 = 6 iterations
		b110 = 7 iterations
		b111 = 8 iterations.

[7]	INV	Input exception flag. Set if the VFP11 coprocessor is not in flush-to-zero mode and an operand is subnormal or if the VFP11 coprocessor is not in default NaN mode and an operand is a NaN.
[6:4]	-	Should Be Zero.
[3]	UFC	Potential underflow flag. Set if the VFP11 coprocessor is not in flush-to-zero mode and a potential underflow condition exists.

Table 17-8 Encoding of the Floating-Point Exception Register (continued)

Bit	Name	Description
[2]	OFC	Potential overflow flag. Set if the OFE bit, FPSCR[10], is set, the VFP11 coprocessor is not in RunFast mode, and a potential overflow condition exists.
[1]	-	Should Be Zero.
[0]	IOC	Potential invalid operation flag. Set if the IOE bit, FPSCR[8], is set, the VFP11 coprocessor is not in RunFast mode, and a potential invalid operation condition exists.

17.4.4 Instruction registers, FPINST and FPINST2

The VFP11 coprocessor has two instruction registers:

- The FPINST register contains the exceptional instruction.
- The FPINST2 register contains the instruction that was issued to the VFP11 coprocessor before the exception was detected. This instruction was retired in the MPCore processor and cannot be reissued. It must be executed by support code.

The FPINST and FPINST2 are accessible only in privileged modes.

The instruction in the FPINST register is in the same format as the issued instruction but is modified in several ways. The condition code flags, FPINST[31:28], are forced to b1110, the AL (always) condition. If the instruction is a short vector, the source and destination registers that reference vectors are updated to point to the source and destination registers of the exceptional iteration. See *Underflow exception* on page 19-21 for more information.

The instruction in the FPINST2 register is in the same format as the issued instruction but is modified by forcing the condition code flags, FPINST2[31:28] to b1110, the AL (always) condition.

17.4.5 Media and VFP Feature Register 0

The purpose of the Media and VFP Feature Register 0 is to provide information about the features that the VFP unit contains.

Media and VFP Feature Register 0 is:

- a 32-bit read-only register

- accessible in any mode when the VFP is enabled by the EN bit, see *Floating-point exception register, FPEXC* on page 17-22
- accessible only in Privileged modes when the VFP is disabled by the EN bit.

Figure 17-8 shows the bit arrangement for Media and VFP Feature Register 0.

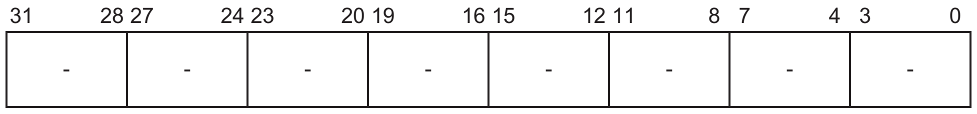


Figure 17-8 Media and VFP Feature Register 0 format

Table 17-9 shows how the bit values correspond with the Media and VFP Feature Register 0 functions.

Table 17-9 Media and VFP Feature Register 0 bit functions

Bit range	Field name	Function
[31:28]	-	Indicates the VFP hardware support level when user traps are disabled. `0x1`, In MPCore processors when Flush-to-Zero and Default_NaN and Round-to-Nearest are all selected in FPSCR, the coprocessor does not require support code. Otherwise floating point support code is required.
[27:24]	-	Indicates support for short vectors. `0x1`, MPCore processors support short vectors.
[23:20]	-	Indicates support for hardware square root. `0x1`, MPCore processors support hardware square root.
[19:16]	-	Indicates support for hardware divide. `0x1`, MPCore processors support hardware divide.
[15:12]	-	Indicates support for user traps. `0x1`, MPCore processors support software traps, support code is required.
[11:8]	-	Indicates support for double precision VFP. `0x1`, MPCore processors support v2.
[7:4]	-	Indicates support for single precision VFP. `0x1`, MPCore processors support v2.
[3:0]	-	Indicates support for the media register bank. `0x1`, MPCore processors support 16, 64-bit registers.

The values in the Media and VFP Feature Register 0 are implementation defined.

17.4.6 Media and VFP Feature Register 1

The purpose of the Media and VFP Feature Register 1 is to provide information about the features that the VFP unit contains.

Media and VFP Feature Register 1 is:

- a 32-bit read-only register
- accessible in any mode when the VFP is enabled by the EN bit, see *Floating-point exception register, FPEXC* on page 17-22
- accessible only in Privileged modes when the VFP is disabled by the EN bit.

Figure 17-9 shows the bit arrangement for Media and VFP Feature Register 1.

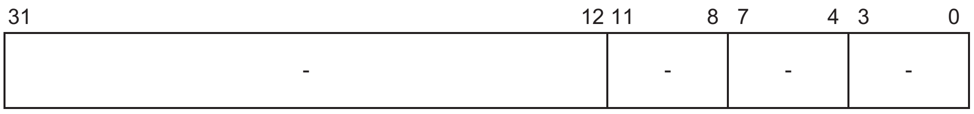


Figure 17-9 Media and VFP Feature Register 1 format

Table 17-10 shows how the bit values correspond with the Media and VFP Feature Register 1 functions.

Table 17-10 Media and VFP Feature Register 1 bit functions

Bit range	Field name	Function
[31:28]	-	Reserved UNP/SBZ.
[11:8]	-	Indicates support for media extension, single precision floating point instructions. `0x0`, no support in ARM11 MPCore processors.
[7:4]	-	Indicates support for media extension, integer instructions. `0x0`, no support in ARM11 MPCore processors.
[3:0]	-	Indicates support for media extension, load/store instructions. `0x0`, no support in ARM11 MPCore processors.

The values in the Media and VFP Feature Register 1 are implementation defined.

Chapter 18
VFP Instruction Execution

This chapter describes the VFP11 instruction pipeline and its relationship with the ARM processor instruction pipeline. It contains the following sections:

18.1 About instruction execution

Features of the VFP11 implementation of the instruction pipelines include the following:

- The FMXR, FMRX, and FMSTAT instructions stall in the VFP11 LS pipeline until all currently executing instructions are completed. You can use these *serializing* instructions to:
  - capture condition codes and exception status
  - modify the mode of operation of subsequent instructions
  - create an exception boundary.

  See *Serializing instructions* on page 18-3.

- Load or store instructions that cause a Data Abort exception restart after interrupt service. LDM and STM instructions detect exceptional conditions after the first transfer and restart after interrupt service if reissued.

  See *Interrupting the VFP11 coprocessor* on page 18-4.

- To reduce stall time, the VFP11 coprocessor forwards data:
  - from load instructions to CDP instructions
  - from CDP instructions to CDP instructions.

  See *Forwarding* on page 18-5.

- In full-compliance mode, the VFP11 coprocessor implements full data hazard and resource hazard detection.

  RunFast mode guarantees no instruction bouncing for applications that require less strict hazard detection.

  See *Hazards* on page 18-7 and *Operation of the scoreboards* on page 18-8.

- The L/S, FMAC, and DS pipelines operate independently, enabling data transfer and CDP operations to execute in parallel.

  See *Parallel execution* on page 18-23.

Execution timing on page 18-25 describes VFP11 instruction throughput and latency.

18.2 Serializing instructions

A serializing instruction is one that stalls due to activity in the VFP11 pipelines without the presence of a register or resource hazard. In general, an access to a VFP11 control or status register is a serializing instruction.

The serializing instructions are FMRX and FMXR, including the FMSTAT instruction. Serializing instructions stall the VFP11 coprocessor in the Issue stage and the ARM processor in the Execute 2 stage until:

- the VFP11 pipeline is past the point of updating either the condition codes or the exception status
- a write to a system register can no longer affect the operation of a current or pending instruction.

An FMRX or FMSTAT instruction stalls until all prior floating-point operations are completed, and the data to be written by the VFP11 coprocessor is valid. For example, a compare operation updates the FPSCR register condition codes in the Writeback stage of the compare.

An FMXR instruction stalls until all prior floating-point operations are past the point of being affected by the instruction. For example, writing to the FPSCR register stalls until the point when changing the control bits cannot affect any operation currently executing or awaiting execution. Writing to the FPEXC, FPINST, or FPINST2 register stalls until the pipeline is completely clear.

Uses of serializing instructions include:

- capturing condition codes and exception status
- delineating a block of instructions for execution with the ability to capture the exception status of that block of instructions
- modifying the mode of operation of subsequent instructions, such as the rounding mode or vector length.

While no instruction can change the contents of the FPSID register, you can access the FPSID register with FMRX or FMXR as a general-purpose serializing operation or to create an exception boundary.

18.3 Interrupting the VFP11 coprocessor

Instructions are issued to the VFP11 coprocessor directly from the ARM prefetch unit. The VFP11 coprocessor has no external interface beyond the ARM processor and cannot be separately interrupted by external sources. Any interrupt that causes a change of flow in the MPCore processor is also reflected to the VFP11 coprocessor. Any VFP instruction that is cancelled due to condition code failure in the MPCore pipeline is also cancelled in the VFP11 pipeline.

If the interrupt is the result of a Data Abort condition, the load or store operation that caused the abort restarts after interrupt processing is complete. Load and store multiple instructions can detect some exception conditions and interrupt the operation after the initial transfer. If the load or store instruction is reissued after interrupt processing, it can restart with the initial transfer. The source data is guaranteed to be unchanged, and no operations that depend on the load or store data can execute until the load or store operation is complete.

When interrupt processing begins, there can be a delay before the VFP11 coprocessor is available to the interrupt routine. Any prior short vector instruction that passes the MPCore Execute 2 stage also passes the VFP11 Execute 1 stage and executes to completion uninterrupted. The maximum delay during which the VFP11 coprocessor is unavailable is equal to the time it takes to process a short vector of eight single-precision divide or square root iterations. Such an operation can cause a delay of as many as 114 cycles after the short vector divide or square root enters the VFP11 Execute 1 stage.

In systems that require fast response time and access to the VFP11 coprocessor by the service routine, avoid short vector divide and short vector square root operations. All other instructions, including short vector instructions, have little or no impact. Limiting the number of VFP11 registers that must be saved and used in the service routine also reduces startup time. If the VFP11 coprocessor is not required in the service routine, you can disable it with EN bit (FPEXC[30]). This eliminates the necessity of saving the VFP11 coprocessor state. See *Application Note 98, VFP Support Code*.

18.4 Forwarding

In general, any forwarding operation reduces the stall time of a dependent instruction by one cycle. The VFP11 coprocessor forwards data from load instructions to CDP instructions and from CDP instructions to CDP instructions.

The VFP11 coprocessor does not forward in the following cases:

- from an instruction that produces integer data
- to a store instruction (FST, FSTM, MRC, or MRRC)
- to an instruction of different precision.

In the examples that follow, the stall counts given are based on two data transfer assumptions:

- accesses by load operations result in cache hits and are able to deliver one or two data words per cycle
- store operations write directly to the write buffer or cache and can transfer one or two data words per cycle.

When these assumptions are valid, the VFP11 coprocessor operates at its highest performance. When these assumptions are not valid, load and store operations are affected by the delay required to access data. The examples below illustrate the capabilities of the VFP11 coprocessor in ideal conditions.

In Example 18-1, the second FADDS instruction depends on the result of the first FADDS instruction. The result of the first FADDS instruction is forwarded, reducing the stall from eight cycles to seven cycles.

Example 18-1 Data forwarded to dependent instruction

```
FADDS S1, S2, S3
FADDS S8, S9, S1
```

In Example 18-2, there is no data forwarding of the double-precision FMULD data in D2 to the single-precision FADDS data in S5, even though S5 is the upper half of D2.

Example 18-2 Mixed-precision data not forwarded

```
FMULD D2, D0, D1
FADDS S12, S13, S5
```

In Example 18-3, the double-precision FSTD stalls for eight cycles until the result of the FMULD is written to the register file. No forwarding is done from the FMULD to the store instruction.

Example 18-3 Data not forwarded to store instruction

```
FMULD D1, D2, D3
FSTD D1, [Rx]
```

18.5 Hazards

The VFP11 coprocessor incorporates full hazard detection with a fully-interlocked pipeline protocol. No compiler scheduling is required to avoid hazard conditions. The source and destination scoreboards process interlocks caused by unavailable source or destination registers or by unavailable data. The scoreboards stall instructions until all data operands and destination registers are available before the instruction is issued to the instruction pipeline.

The determination of hazards and interlock conditions is different in full-compliance mode and RunFast mode. RunFast mode guarantees no bounce conditions and has a less strict hazard detection mechanism, enabling instructions to begin execution earlier than in full-compliance mode.

There are two VFP11 pipeline hazards:

- A data hazard is a combination of instructions that creates the potential for operands to be accessed in the wrong order.
  - A *Read-After-Write* (RAW) data hazard occurs when the pipeline creates the potential for an instruction to read an operand before a prior instruction writes to it. It is a hazard to the intended read-after-write operand access.
  - A *Write-After-Read* (WAR) data hazard occurs when the pipeline creates the potential for an instruction to write to a register before a prior instruction reads it. It is a hazard to the intended write-after-read operand access.
  - A *Write-After-Write* (WAW) data hazard occurs when the pipeline creates the potential for an instruction to write to a register before a prior instruction writes to it. It is a hazard to the intended write-after-write operand access.
- Resource hazard. See *Resource hazards* on page 18-20.

18.6 Operation of the scoreboards

The VFP11 processor detects all hazard conditions that exist between issued and executing instructions. It uses two scoreboards to ensure that all source and destination registers for an instruction contain valid data and are available for reading or writing:

- The destination scoreboard contains a lock for each destination register for the current operation.
- The source scoreboard contains a lock for each source register for the current operation.

In the Decode stage of the VFP11 pipeline, the VFP11 coprocessor determines which source and destination registers are involved in an operation and generates a lock mask for them. In a short vector operation, the lock mask includes the registers involved in every iteration of the operation. In the Issue stage, the VFP11 coprocessor checks and updates the source and destination scoreboards. If it detects a hazard between the instruction in the Issue stage and a prior instruction, the scoreboards are not updated, and the instruction stalls in the Issue stage.

A VFP11 instruction can begin execution only when its source and destination registers are free of locks. A short vector operation can begin only when the registers for all its iterations are free of locks. When a short vector instruction proceeds in the pipeline beyond the Issue stage, all the registers involved in the operation are locked.

The source scoreboard clears a source register lock in the first Execute 1 stage of the pipeline or in the first Execute 1 stage of the iteration. In store multiple instructions, the source scoreboard clears source register locks in the Execute stage in which the instruction writes the store data to the MPCore processor.

The destination scoreboard clears the destination register lock in the cycle before the result data is written back to the register file or is available for forwarding (Execute 7 in the FMAC pipeline, Execute 4 in the DS pipeline). In a load operation, the destination scoreboard normally clears the destination register lock in the Memory 2 stage. If the load is delayed, the destination scoreboard clears the destination register lock in the same cycle as the writeback to the register file.

18.6.1 Scoreboard operation when an instruction bounces

When a bounce occurs in full-compliance mode, support code is called to complete the operation and to deliver the result and the exception status to the user trap handler. The source scoreboard ensures that all source registers for the operation are preserved for the support code. In a short vector operation, this includes the source registers for the bounced iteration and for any iterations remaining after the bounced iteration. The preserved source registers include the destination register for a multiply and accumulate instruction.

Because RunFast mode guarantees that no bouncing is possible, source registers do not have to be preserved after they are used by the instruction. For all scalar operations and nonmultiple store operations, no source registers are locked in RunFast mode. In short vector operations, the length of the vector determines which source registers are locked. When the vector length exceeds four single-precision iterations, the source scoreboard locks the source registers for iterations 5 and above. When the vector length exceeds two double-precision iterations, the source scoreboard locks the source registers for iterations 3 and above.

18.6.2 Single-precision source register locking

In full-compliance mode, the source scoreboard locks all source registers in the Issue stage of the instruction. In RunFast mode, the source scoreboard locks the source registers for only iterations 5, 6, 7, and 8. Table 18-1 summarizes source register locking in single-precision operations.

Table 18-1 Single-precision source register locking

		Source registers locked in Issue stage	
LEN	**Vector length**	**Full-compliance mode**	**RunFast mode**
b000	1	Iteration 1 registers	-
b001	2	Iteration 1-2 registers	-
b010	3	Iteration 1-3 registers	-
b011	4	Iteration 1-4 registers	-
b100	5	Iteration 1-5 registers	Iteration 5 registers
b101	6	Iteration 1-6 registers	Iteration 5-6 registers
b110	7	Iteration 1-7 registers	Iteration 5-7 registers
b111	8	Iteration 1-8 registers	Iteration 5-8 registers

For the following single-precision short vector instruction, the LEN field contains b100, selecting a vector length of five iterations:

```
FADDS S8, S16, S24
```

The FADDS instruction performs the following operations:

```
FADDS S8, S16, S24
FADDS S9, S17, S25
FADDS S10, S18, S26
FADDS S11, S19, S27
FADDS S12, S20, S28
```

In full-compliance mode, the source scoreboard locks S16-S20 and S24-S28 in the Issue stage of the instruction.

In RunFast mode, the source scoreboard locks only the fifth iteration source registers, S20 and S28.

18.6.3 Single-precision source register clearing

In full-compliance mode, the source scoreboard clears the source registers of each iteration in the Execute 1 stage of the iteration. In RunFast mode, the source registers for only iterations 5, 6, 7, and 8 are locked, and the source scoreboard begins clearing them in the second Execute 1 cycle of the instruction. Table 18-2 summarizes source register clearing in single-precision operations.

Table 18-2 Single-precision source register clearing

	Source registers cleared in Execute 1 stage of each iteration	
Execute 1 cycle	**Full-compliance mode**	**RunFast mode**
1	Iteration 1 registers	-
2	Iteration 2 registers	Iteration 5 registers
3	Iteration 3 registers	Iteration 6 registers
4	Iteration 4 registers	Iteration 7 registers
5	Iteration 5 registers	Iteration 8 registers
6	Iteration 6 registers	-
7	Iteration 7 registers	-
8	Iteration 8 registers	-

For the following single-precision short vector instruction, the LEN field contains b100, selecting a vector length of five iterations:

```
FADDS S8, S16, S24
```

The FADDS instruction performs the following operations:

```
FADDS S8, S16, S24
FADDS S9, S17, S25
FADDS S10, S18, S26
FADDS S11, S19, S27
FADDS S12, S20, S28
```

In full-compliance mode, the source scoreboard clears the source registers of each iteration in the Execute 1 cycle of the iteration.

In RunFast mode, the source scoreboard locks only the fifth iteration source registers, S20 and S28. It clears S20 and S28 in the second Execute 1 cycle of the instruction.

18.6.4 Double-precision source register locking

In full-compliance mode, the source scoreboard locks all source registers in the Issue stage of the instruction. In RunFast mode, the source scoreboard locks the source registers for only iterations 3 and 4. Table 18-3 summarizes source register locking in double-precision operations.

Table 18-3 Double-precision source register locking

		Source registers locked in Issue stage	
LEN	**Vector length**	**Full-compliance mode**	**RunFast mode**
b000	1	Iteration 1 registers	-
b001	2	Iteration 1-2 registers	-
b010	3	Iteration 1-3 registers	Iteration 3 registers
b011	4	Iteration 1-4 registers	Iteration 3-4 registers

For the following double-precision, short vector instruction, the LEN field contains b011, selecting a vector length of four iterations:

```
FADDD D4, D8, D12
```

The FADDD instruction performs the following operations:

```
FADDD D4, D8, D12
FADDD D5, D9, D13
FADDD D6, D10, D14
FADDD D7, D11, D15
```

In full-compliance mode, the source scoreboard locks D8-D11 and D12-D15 in the Issue stage of the instruction.

In RunFast mode, the source scoreboard locks only the third iteration source registers, D10 and D14, and the fourth iteration source registers, D11 and D15.

18.6.5 Double-precision source register clearing

The number of Execute 1 cycles required to clear the source registers of a double-precision instruction depends on the throughput of the instruction, as the following sections show:

- *Instructions with one-cycle throughput*
- *Instructions with two-cycle throughput* on page 18-13.

Instructions with one-cycle throughput

In full-compliance mode, the source scoreboard clears the source registers of each iteration in the Execute 1 stage of the iteration. In RunFast mode, the source registers for only iterations 3 and 4 are locked, and the source scoreboard begins clearing them in the first Execute 1 cycle of the instruction. Table 18-4 summarizes source register clearing for double-precision one-cycle instructions such as FADDD and FABSD.

Table 18-4 Double-precision source register clearing for one-cycle instructions

Execute 1 cycle	Source registers cleared in Execute 1 stage of each iteration	
	Full-compliance mode	**RunFast mode**
1	Iteration 1 registers	Iteration 3 registers
2	Iteration 2 registers	Iteration 4 registers
3	Iteration 3 registers	-
4	Iteration 4 registers	-

For the following one-cycle, double-precision short vector instruction, the LEN field contains b011, selecting a vector length of four iterations:

```
FADDD D4, D8, D12
```

The FADDD performs the following operations:

```
FADDD D4, D8, D12
FADDD D5, D9, D13
FADDD D6, D10, D14
FADDD D7, D11, D15
```

In full-compliance mode, the source scoreboard clears the source registers of each iteration in the Execute 1 cycle of the iteration.

In RunFast mode, the source scoreboard locks only the third iteration source registers, D10 and D14, and the fourth iteration source registers, D11 and D15. It clears D10 and D14 in the first Execute 1 cycle of the instruction and clears D11 and D15 in the second Execute 1 cycle.

Instructions with two-cycle throughput

In full-compliance mode, the source scoreboard clears the source registers of each iteration in the first Execute 1 cycle of the iteration. In RunFast mode, the source registers for only iterations 3 and 4 are locked, and the source scoreboard begins clearing them in the first Execute 1 cycle of the instruction. Table 18-5 summarizes source register clearing for double-precision two-cycle instructions such as FMULD and FMACD.

Table 18-5 Double-precision source register clearing for two-cycle instructions

Execute 1 cycle	Source registers cleared in Execute 1 stage of each iteration	
	Full-compliance mode	**RunFast mode**
1	Iteration 1 registers	Iteration 3 registers
2	-	-
3	Iteration 2 registers	Iteration 4 registers
4	-	-
5	Iteration 3 registers	-
6	-	-
7	Iteration 4 registers	-
8	-	-

For the following two-cycle, double-precision, short vector instruction, the LEN field contains b011, selecting a vector length of four iterations:

```
FMULD D4, D8, D12
```

The FMULD instruction performs the following operations:

```
FMULD D4, D8, D12
FMULD D5, D9, D13
FMULD D6, D10, D14
FMULD D7, D11, D15
```

In full-compliance mode, the source scoreboard clears the source registers of each iteration in the first Execute 1 cycle of the iteration.

In RunFast mode, only the third iteration source registers, D10 and D14, and the fourth iteration source registers, D11 and D15, are locked. The source scoreboard clears D10 and D14 in the first Execute 1 cycle and clears D11 and D15 in the third Execute 1 cycle of the instruction.

18.7 Data hazards in full-compliance mode

The sections that follow give examples of data hazards in full-compliance mode:

- *Status register RAW hazard example*
- *Load multiple-CDP RAW hazard example*
- *CDP-CDP RAW hazard example* on page 18-17
- *Load multiple-short vector CDP RAW hazard example* on page 18-16
- *Short vector CDP-load multiple WAR hazard example* on page 18-17.

18.7.1 Status register RAW hazard example

In Example 18-4, the FMSTAT is stalled for four cycles in the Decode stage until the FCMPS updates the condition codes in the FPSCR register. Two cycles later, the FMSTAT writes the condition codes to the MPCore processor.

Example 18-4 FCMPS-FMSTAT RAW hazard

```
FCMPS S1, S2
FMSTAT
```

Table 18-6 shows the VFP11 pipeline stages for Example 18-4.

Table 18-6 FCMPS-FMSTAT RAW hazard

	Instruction cycle number										
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
FCMPS	D	I	E1	E2	E3	E4	-	-	-	-	-
FMSTAT	-	D	D	D	D	D	I	E	M1	M2	W

18.7.2 Load multiple-CDP RAW hazard example

In Example 18-5, the FADDS is stalled in the Issue stage for six cycles until the FLDM makes its last transfer to the VFP11 coprocessor. S15 is forwarded from the load in cycle 9 to the FADDS.

Example 18-5 FLDM-FADDS RAW hazard

```
FLDM [Rx], {S8-S15}
```

```
FADDS S1, S2, S15
```

Table 18-7 shows the VFP11 pipeline stages for Example 18-5 on page 18-15.

Table 18-7 FLDM-FADDS RAW hazard

	Instruction cycle number															
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
FLDM	D	I	E	M1	M2	W	W	W	W	-	-	-	-	-	-	-
FADDS	-	D	I	I	I	I	I	I	I	E1	E2	E3	E4	E5	E6	E7

18.7.3 Load multiple-short vector CDP RAW hazard example

In Example 18-6, the short vector FADDS is stalled in the Issue stage until the FLDM loads all source registers required by the FADDS. In this case, the FADDS is stalled for three cycles. Because the FADDS depends on the FLDM for only one register, S7, it does not have to wait for completion of the FLDM. The S7 data is forwarded in cycle 6. The LEN field contains b011, selecting a vector length of four iterations. The STRIDE field contains b00, selecting a vector stride of one. The first source vector uses registers S7, S0, S1, and S2, and the only FADDS source register loaded by the FLDM is S7. This example is based on the assumption that the remaining source and destination registers are available to the FADDS in cycle 6.

Example 18-6 FLDM-short vector FADDS RAW hazard

```
FLDM [R2], {S7-S14}
FADDS S16, S7, S25
```

Table 18-8 shows the VFP11 pipeline stages of the FLDM and the first iteration of the short vector FADDS for Example 18-6.

Table 18-8 FLDM-short vector FADDS RAW hazard

	Instruction cycle number																
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
FLDM	D	I	E	M1	M2	W	W	W	W	-	-	-	-	-	-	-	-
FADDS	-	D	I	I	I	I	E1	E1	E1	E1	E2	E3	E4	E5	E6	E7	W

18.7.4 CDP-CDP RAW hazard example

In Example 18-7, the FADDS is stalled in the Issue stage for seven cycles until the FMULS data is written and forwarded in cycle 10 to the Issue stage of the FADDS.

Example 18-7 FMULS-FADDS RAW hazard

```
FMULS S4, S1, S0
FADDS S5, S4, S3
```

Table 18-9 shows the VFP11 pipeline stages of Example 18-7.

Table 18-9 FMULS-FADDS RAW hazard

	Instruction cycle number										
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
FMULS	D	I	E1	E2	E3	E4	E5	E6	E7	W	-
FADDS	-	D	I	I	I	I	I	I	I	I	EI

18.7.5 Short vector CDP-load multiple WAR hazard example

In Example 18-8, the load multiple FLDMS creates a WAR hazard to the source registers of the FMULS. The LEN field contains b011, selecting a vector length of four iterations, and the STRIDE field contains b00, selecting a vector stride of one. The VFP11 coprocessor stalls the FLDMS until the FMULS clears the scoreboard locks for all the source registers, S16-S19 and S24-S27.

Example 18-8 Short vector FMULS-FLDMS WAR hazard

```
FMULS S8, S16, S24
FLDMS [R2], {S16-S27}
```

Table 18-10 on page 18-18 shows the VFP11 pipeline stages for the first iteration of Example 18-8.

Table 18-10 Short vector FMULS-FLDMS WAR hazard

	Instruction cycle number															
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
FMULS	D	I	E1	E1	E1	E1	E2	E3	E4	E5	E6	E7	W	-	-	-
FLDMS	-	D	I	I	I	I	I	E	M1	M2	W	W	W	W	W	W

18.8 Data hazards in RunFast mode

In RunFast mode, source registers for the FMAC and FMUL family of instructions are locked:

- when the vector length exceeds four iterations in single-precision instructions
- when the vector length exceeds two iterations in double-precision instructions.

No source registers are locked for scalar instructions.

18.8.1 Short vector CDP-load multiple WAR hazard example

Example 18-9 is the same as Example 18-8 on page 18-17. The LEN field contains b011, selecting a vector length of four iterations, and the STRIDE field contains b00, selecting a vector stride of one. Executing these instructions in RunFast mode reduces the cycle count of the FLDMS by four cycles.

Example 18-9 Short vector FMULS-FLDMS WAR hazard in RunFast mode

```
FMULS S8, S16, S24
FLDMS R2, {S16-S27}
```

Table 18-11 shows that the VFP11 coprocessor does not stall the FLDMS operation.

Table 18-11 Short vector FMULS-FLDMS WAR hazard in RunFast mode

	Instruction cycle number												
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
FMULS	D	I	E1	E1	E1	E1	E2	E3	E4	E5	E6	E7	W
FLDMS	-	D	I	E	M1	M2	W	W	W	W	W	W	-

18.9 Resource hazards

A resource hazard exists when the pipeline required for an instruction is unavailable due to a prior instruction. VFP11 resource stalls are possible in the following cases:

- A data transfer operation following an incomplete data transfer operation can cause a resource stall. The MPCore processor can stall each data transfer because of unavailable data caused by memory latency or a cache miss, increasing the latency of the data transfer instruction and stalling any following data transfer instructions.
- An arithmetic operation following either a short vector arithmetic operation or a double-precision multiply or multiply and accumulate operation can cause a resource stall. The latency for a double-precision multiply or multiply and accumulate operation is two cycles, causing a single-cycle stall for an arithmetic operation that immediately follows.
- A single-precision divide or square root operation stalls subsequent DS operations for 15 cycles. A double-precision divide or square root operation stalls subsequent DS operations for 29 cycles.
- A short vector divide or square root operation requires the FMAC pipeline for the first cycle of each iteration and stalls any following CDP operation. The following CDP operation stalls until the final iteration of the short vector divide or square root operation completes the Execute 1 stage.

The LS pipeline is separate from the FMAC and DS pipelines. No resource hazards exist between data transfer instructions and arithmetic instructions.

The sections that follow give examples of resource hazards:

- *Load multiple-load-CDP resource hazard example*
- *Load multiple-short vector CDP resource hazard example* on page 18-21
- *Short vector CDP-CDP resource hazard example* on page 18-22.

18.9.1 Load multiple-load-CDP resource hazard example

In Example 18-10, the FLDM is executing two transfers to the VFP11 coprocessor. The FLDS is stalled behind the FLDM until the FLDM enters the final Execute cycle. The FADDS is stalled for one cycle until the FLDS begins execution.

Example 18-10 FLDM-FLDS-FADDS resource hazard

```
FLDM [R2], {S8-S10}
FLDS [R4], S16
```

```
FADDS S2, S3, S4
```

Table 18-12 shows the pipeline stages for Example 18-10 on page 18-20.

Table 18-12 FLDM-FLDS-FADDS resource hazard

	Instruction cycle number												
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
FLDM	D	I	E	M1	M2	W	W	-	-				
FLDS	-	D	D	I	E	M1	M2	W	-				
FADDS	-	-	-	D	I	E1	E2	E3	E4	E5	E6	E7	W

18.9.2 Load multiple-short vector CDP resource hazard example

In Example 18-11, no resource hazard exists for the FMULS due to the FLDM in the prior cycle. The FMULS is issued to the VFP11 coprocessor in the cycle following the issue of the FLDM, and executes in parallel with it.

The LEN field contains, b011, selecting a vector length of four iterations. The STRIDE field contains b00, selecting a vector stride of one.

Example 18-11 FLDM-short vector FMULS resource hazard

```
FLDM [R2], {S8-S10}
FMULS S16, S24, S4
```

Table 18-13 shows the pipeline stages for Example 18-11.

Table 18-13 FLDM-short vector FMULS resource hazard

	Instruction cycle number													
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
FLDM	D	I	E	M1	M2	W	W	-	-					
FMULS	-	D	I	E1	E1	E1	E1	E2	E3	E4	E5	E6	E7	W

18.9.3 Short vector CDP-CDP resource hazard example

In Example 18-12, a short vector divide is followed by a FADDS instruction. The short vector divide has b001 in the LEN field, selecting a vector length of two iterations. It requires the Execute 1 stage of the FMAC pipeline for the first cycle of each iteration of the divide, resulting in a stall of the FADDS until the final iteration of the divide completes the first Execute 1 cycle. The divide iterates for 14 cycles in the Execute 1 and Execute 2 stages of the DS pipeline, shown in Table 18-14 as E1. The first and shared Execute 1 cycle for each divide iteration is designated as E1'.

Example 18-12 Short vector FDIVS-FADDS resource hazard

```
FDIVS S8, S10, S12
FADDS S0, S0, S1
```

Table 18-14 shows the pipeline stages for Example 18-12.

Table 18-14 Short vector FDIVS-FADDS resource hazard

	Instruction cycle number																						
Instruction	**1**	**2**	**3**	**4**	**...**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**...**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
FDIVS	D	I	E1'	E1	...	E1	E1	E1'	E1	E1	E1	E1	E1	E1	E1	E1	...	E1	E1	E1	E2	E3	E4	W
FADDS	-	-	D	D	...	D	D	I	E1	E2	E3	E4	E5	E6	E7	W	...	-	-	-	-	-	-	-

18.10 Parallel execution

The VFP11 coprocessor is capable of execution in each of the three pipelines independently of the others and without blocking issue or writeback from any pipeline. Separate LS, FMAC, and DS pipelines allow for parallel operation of CDP and data transfer instructions. Scheduling instructions to take advantage of the parallelism that occurs when multiple instructions execute in the VFP11 pipelines can result in a significant improvement in program execution time.

A data transfer operation can begin execution if:

- no data hazards exist with any currently executing operations
- the LS pipeline is not currently stalled by the MPCore processor or busy with a data transfer multiple.

A CDP can be issued to the FMAC pipeline if:

- no data hazards exist with any currently executing operations
- the FMAC pipeline is available (no short vector CDP is executing and no double-precision multiply is in the first cycle of the multiply operation)
- no short vector operation with unissued iterations is currently executing in either the FMAC or DS pipeline.

A divide or square root instruction can be issued to the DS pipeline if:

- no data hazards exist with any currently executing operations
- the DS pipeline is available (no current divide or square root is executing in the DS pipeline E1 stage)
- no short vector operation with unissued iterations is executing in the FMAC pipeline.

Example 18-13 on page 18-24 shows a case of the VFP11 coprocessor executing instructions in parallel in each of the three pipelines:

- a load multiple in the L/S pipeline
- a divide in the DS pipeline
- a short vector add in the FMAC pipeline.

In this example, the LEN field contains b011, selecting a vector length of four iterations, and the STRIDE field contains b00, for a vector stride of one.

Example 18-13 Parallel execution in all three pipelines

```
FLDM [R4], {S4-S13}
FDIVSS0, S1, S2
FADDS S16, S20, S24
```

Table 18-15 shows the pipeline progression of the three instructions.

Table 18-15 Parallel execution in all three pipelines

	Instruction cycle number														
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
FLDM	D	I	E	M1	M2	W	W	W	W	W	-	-	-	-	-
FDIVS	-	D	I	E1’	E1	E1	E1	E1	E1	E1	E1	E1	E1	E1	E1
FADDS	-	-	D	I	E1	E1	E1	E1	E2	E3	E4	E5	E6	E7	W

In Example 18-13, no data hazards exist between any of the three instructions. The load multiple is able to begin execution immediately, and data is transferred to the register file beginning in cycle 6. Because the destination is in bank 0, the FDIVS is a scalar operation and requires one cycle in the FMAC pipeline E1 stage. If the FDIVS were a short vector operation, the FADDS could not begin execution until the last FDIVS iteration passed the FMAC E1 pipeline stage. The FADDS is a short vector operation and requires the FMAC pipeline E1 stage for cycles 5-8.

Note

E1’ is the first cycle in E1 and is in both FMAC and DS blocks. Subsequent E1 cycles represent the iteration cycles and occupy both E1 and E2 stages in the DS block.

18.11 Execution timing

Complex instruction dependencies and memory system interactions make it impossible to describe briefly the exact cycle timing of all instructions in all circumstances. The timing described in Table 18-16 is accurate in most cases. For precise timing, you must use a cycle-accurate model of your MPCore processor.

In Table 18-16, throughput is defined as the cycle after issue in which another instruction can begin execution. Instruction latency is the number of cycles after which the data is available for another operation. Forwarding reduces the latency by one cycle for operations that depend on floating-point data. Table 18-16 shows the throughput and latency for all VFP11 instructions.

Table 18-16 Throughput and latency cycle counts for VFP11 instructions

	Single-precision		Double-precision	
Instructions	**Throughput**	**Latency**	**Throughput**	**Latency**
FABS, FNEG, FCVT, FCPY	1	4	1	4
FCMP, FCMPE, FCMPZ, FCMPEZ	1	4	1	4
FSITO, FUITO, FTOSI, FTOUI, FTOUIZ, FTOSIZ	1	8	1	8
FADD, FSUB	1	8	1	8
FMUL, FNMUL	1	8	2	9
FMAC, FNMAC, FMSC, FNMSC	1	8	2	9
FDIV, FSQRT	15	19	29	33
FLD[a]	1	4	1	4
FST[a]	1[a]	System-dependent	1	System-dependent
FLDM[a]	X[b]	X[b] + 3	X[b]	X[b] + 3
FSTM[a]	X[b]	System-dependent	X[b]	System-dependent
FMSTAT	1	2	-	-
FMSR/FMSRR[c]	1	4	-	-
FMDHR/FMDHC/FMDRR[c]	-	-	1	4
FMRS/FMRRS[c]	1	2	-	-

Table 18-16 Throughput and latency cycle counts for VFP11 instructions (continued)

	Single-precision		Double-precision	
Instructions	**Throughput**	**Latency**	**Throughput**	**Latency**
FMRDH/FMRDL/FMRRD[c]	-	-	1	2
FMXR[d]	1	4	-	-
FMRX[d]	1	2	-	-

a. The cycle count for a load instruction is based on load data that is cached and available to the MPCore processor from the cache. The cycle count for a store instruction is based on store data that is written to the cache and/or write buffer immediately. When the data is not cached or the write buffer is unavailable, the number of cycles also depends on the memory subsystem.

b. The number of cycles represented by X is (N/2) if N is even or (N/2 + 1) if N is odd.

c. FMDRR and FMRRD transfer one double-precision data per transfer. FMSRR and FMRRS transfer two single-precision data per transfer.

d. FMXR and FMRX are serializing instructions. The latency depends on the register transferred and the current activity in the VFP11 coprocessor when the instruction is issued.

Chapter 19
VFP Exception Handling

This chapter describes VFP11 exception processing. It contains the following sections:

19.1 About exception processing

The VFP11 coprocessor handles exceptions, other than inexact exceptions, imprecisely with respect to both the state of the MPCore processor and the state of the VFP11 coprocessor. It detects an exceptional instruction after the instruction passes the point for exception handling in the MPCore processor. It then enters the *exceptional state* and signals the presence of an exception by refusing to accept a subsequent VFP instruction. The instruction that triggers exception handling bounces to the MPCore processor. The bounced instruction is not necessarily the instruction immediately following the exceptional instruction. Depending on sequence of instructions that follow, the bounce can occur several instructions later.

The VFP11 coprocessor can generate exceptions only on arithmetic operations. Data transfer operations between the MPCore processor and the VFP11 coprocessor, and instructions that copy data between VFP11 registers, FCPY, FABS, and FNEG, cannot produce exceptions.

In full-compliance mode the VFP11 hardware and support code together process exceptions according to the IEEE 754 standard. VFP11 exception processing includes calling user trap handlers with intermediate operands specified by the IEEE 754 standard. In RunFast mode, the VFP11 coprocessor generates the default, or trap disabled, value when an overflow, invalid operation, division by zero, or inexact condition occurs. RunFast mode does not provide for user trap handlers.

For descriptions of each of the exception flags and their bounce characteristics, see the sections *Input Subnormal exception* on page 19-14 to *Arithmetic exceptions* on page 19-25.

19.2 Bounced instructions

Normally, the VFP11 hardware executes floating-point instructions completely in hardware. However, the VFP11 coprocessor can, under certain circumstances, refuse to accept a floating-point instruction, causing the ARM Undefined Instruction exception. This is known as *bouncing* the instruction.

There are three reasons for bouncing an instruction:

- a prior instruction generates a potential or actual floating-point exception that cannot be properly handled by the VFP11 coprocessor, such as a potential underflow when the VFP11 coprocessor is not in flush-to-zero mode
- a prior instruction generates a potential or actual floating-point exception when the corresponding exception enable bit is set in the FPSCR, such as a square root of a negative value when the IOE bit, FPSCR[8], is set
- the current instruction is Undefined.

When a floating-point exception is detected, the VFP11 hardware sets the EX flag, FPEXC[31], and loads the FPINST register with a copy of the exceptional instruction. The VFP11 coprocessor is now in the *exceptional state*. The instruction that bounces as a result of the exceptional state is referred to as the *trigger* instruction.

See *Exception processing* on page 19-8.

19.2.1 Potential or actual exception that the VFP11 coprocessor cannot handle

Three exceptional conditions cannot be handled by the VFP11 hardware:

- an operation that might underflow when the VFP11 coprocessor is not in flush-to-zero mode
- an operation involving a subnormal operand when the VFP11 coprocessor is not in flush-to-zero mode
- an operation involving a NaN when the VFP11 coprocessor is not in default NaN mode.

For these conditions the VFP11 coprocessor relies on support code to process the operation. See *Underflow exception* on page 19-21 and *Input exceptions* on page 19-24.

19.2.2 Potential or actual exception with the exception enable bit set

The VFP11 coprocessor evaluates the instruction for exceptions in the E1 and E2 pipeline stages. No means exist to signal exceptions to the MPCore processor after the E2 stage. The VFP11 coprocessor enters the exceptional state when it detects that an instruction has a potential to generate a floating-point exception while the corresponding exception enable bit is set. Such an instruction is called a *potentially exceptional instruction*.

An example of an instruction that generates an actual exception is a division of a normal value by zero when the Division by Zero exception enable bit, FPSCR[9], is set. This mechanism provides support for the IEEE 754 trap mechanism and provides programmers a means of halting execution on certain conditions.

As an example of an instruction that generates a potential exception, if the overflow exception enable bit, FPSCR[10], is set, and the initial exponent for a multiply operation is the maximum exponent for a normal value in the destination precision, the VFP11 coprocessor bounces the instruction pessimistically. Since the impact on the exponent due to mantissa overflow and rounding is not known in the E1 or E2 stages of the FMAC pipeline, the decision to bounce must be made based on the potential for an exception. Support code performs the multiply operation and determines the exception status. If the multiply operation results in an overflow, the processor jumps to the Overflow user trap handler. If the operation does not result in an overflow, it writes the computed result to the destination, sets the appropriate flags in the FPSCR, and returns to user code.

19.3 Support code

The VFP11 coprocessor provides floating-point functionality through a combination of hardware and software support.

When an instruction bounces, software installed on the ARM Undefined Instruction vector determines why the VFP11 coprocessor rejected the instruction and takes appropriate remedial action. This software is called the *VFP support code*. The support code has two components:

- a library of routines that perform floating-point arithmetic functions
- a set of exception handlers that process exceptional conditions.

See *Application Note 98, VFP Support Code* for details of support code. Support code is provided with the RealView Compilation Tools, or for the ARM Developer Suite as an add-on downloadable from the ARM web site.

The remedial action is performed as follows:

1. The support code starts by reading the FPEXC register. If the EX flag, FPEXC[31], is set, a potential exception is present. If not, an illegal instruction is detected. See *Illegal instructions* on page 19-6.

   The contents of the FPEXC register must be retained throughout exception processing. Any VFP11 coprocessor activity might change FPEXC register bits from their state at the time of the exception.

2. The support code writes to the FPEXC register to clear the EX flag. Failure to do this can result in an infinite loop of exceptions when the support code next accesses the VFP11 hardware.

3. The support code reads the FPSCR to determine if IXE is set or not set. If IXE, FPSCR[12], is set, an inexact exception has occurred, that takes priority over other exceptions and is precise. Other exceptions are imprecise.

4. The support code reads either the FPINST register, or the instruction pointed to by r14-4, depending on whether the exception is precise or not, to determine the instruction that caused the potential exception.

5. The support code decodes the instruction in the FPINST register, reads its operands, including implicit information such as the rounding mode and vector length in the FPSCR register, executes the operation, and determines whether a floating-point exception occurred.

6. If no floating-point exception occurred, the support code writes the correct result of the operation and sets the appropriate flags in the FPSCR register.

If one or more floating-point exceptions occurred, but all of them were disabled, the support code determines the correct result of the instruction, writes it to the destination register, and sets the corresponding flags in the FPSCR register.

If one or more floating-point exceptions occurred, and at least one of them was enabled, the support code computes the intermediate result specified by the IEEE 754 standard, if required, and calls the user trap handler for that exception. The user trap handler can provide a result for the instruction and continue program execution, generate a signal or message to the operating system or the user, or simply terminate the program.

7. If the potentially exceptional instruction specified a short vector operation, the hardware does not execute any vector iterations after the one that encountered the potentially exceptional condition. The support code repeats steps 4 and 5 for any such iterations. See *Exception processing for CDP short vector instructions* on page 19-9 for more details.
8. If the FP2V flag, FPEXC[28], is set and IXE, FPSCR[12], is clear, the FPINST2 register contains another VFP instruction that was issued between the potentially exceptional instruction and the trigger instruction. This instruction is executed by the support code in the same manner as the instruction in the FPINST register. The FP2V flag must be cleared before returning to user code. See *Instruction registers, FPINST and FPINST2* on page 17-25 for more on FPINST2.
9. The support code finishes processing the potentially exceptional instruction and returns to the program containing the trigger instruction. The MPCore processor refetches the trigger instruction from memory and reissues it to the VFP11 coprocessor. Unless another bounce occurs, the trigger instruction is executed. Returning in this fashion is called *retrying* the trigger instruction.

The support code can be written to use the VFP11 hardware for its internal calculations, provided that:

- recursive bounces are prevented or handled correctly
- care is taken to restore the state of the original program before returning to it.

Restoring the state of the original program can be difficult if the original program was executing in FIQ mode or in Undefined instruction mode. It is legitimate for support code to disallow or restrict the use of VFP11 instructions in these two processor modes.

19.3.1 Illegal instructions

If there is not a potential floating-point exception from an earlier instruction, the current instruction can still be bounced if it is architecturally Undefined in some way. When this happens, the EX flag, FPEXC[31], is not set. The instruction that caused the bounce is contained in the memory word pointed to by r14_undef – 4.

It is possible that both conditions for an instruction to be bounced occur simultaneously. This happens when an illegal instruction is encountered and there is also a potential floating-point exception from an earlier instruction. When this happens, the EX flag is set, and the support code processes the potential exception in the earlier instruction. If and when it returns, it causes the illegal instruction to be retried and the sequence of events described in the paragraph above occurs.

The following instruction types are architecturally Undefined (see *ARM Architecture Reference Manual, Rev E, Part C)*:

- instructions with opcode bit combinations defined as reserved in the architecture specification
- load or store instructions with Undefined P, W, and U bit combinations
- FMRX/FMXR instructions to or from a control register that is not defined
- User mode FMRX/FMXR instructions to or from a control register that can be accessed only in a privileged mode
- double precision operations with odd register numbers.

Certain instruction types do not have architecturally-defined behavior and are Unpredictable:

- load or store multiple instructions with a transfer count of zero or greater than 32, and any combination of initial register and transfer count such that an attempt is made to transfer a register beyond S31 for single-precision transfers, or D15 for double-precision transfers
- a short vector instruction with a combination of precision, length, and stride that causes the vector to wrap around and make more than one access to the same register
- a short vector instruction with overlapping source and destination register addresses that are not exactly the same.

19.4 Exception processing

The MPCore/VFP11 interface specifies that an exceptional instruction that bounces to support code must signal on a subsequent coprocessor instruction. This is known as *imprecise exception handling*. It means that when the exception is processed, the VFP11 and MPCore user states might be different from their states when the exceptional instruction executed. Parallel execution of VFP11 CDP instructions and data transfer instructions allows the VFP11 and MPCore register files and memory to be modified outside of the program order.

19.4.1 Determination of the trigger instruction

The issue timing of VFP11 instructions affects the determination of the trigger instruction. The last iteration of a short vector CDP can be followed in the next cycle by a second CDP instruction. If there is no hazard, the VFP11 coprocessor accepts the second CDP instruction before the exception status of the last iteration of the short vector CDP is known. The second CDP instruction is said to be in the *pretrigger slot* and is retained in the FPINST2 register for the support code.

The following rules determine which instruction is the trigger instruction:

- The first nonserializing instruction after the exceptional condition has been detected is the trigger instruction.
- An instruction that accesses the FPSCR register in any processor mode is a trigger instruction.
- An instruction that accesses the FPEXC, FPINST, or FPINST2 register in a privileged mode is not a trigger instruction.
- An instruction that accesses the FPSID register in any mode is not a trigger instruction.
- A data processing instruction that reaches the LS pipeline Execute stage or a CDP instruction that reaches the FMAC or DS pipeline E1 stage is not the trigger instruction. There can be several of these if the exceptional instruction is a sufficiently long short vector instruction, and the exception is detected on a later iteration.

19.4.2 Exception processing for CDP scalar instructions

When the VFP11 coprocessor detects an exceptional scalar CDP instruction, it loads the FPINST register with the instruction word for the exceptional instruction and flags the condition in the FPEXC register. It blocks the exceptional instruction from further execution and completes any instructions currently executing in the FMAC and DS pipelines.

It then examines the pipeline for a trigger instruction:

- If there is a VFP CDP instruction or a load or store instruction in the VFP11 Issue stage, it is the trigger instruction and is bounced in the cycle after the exception is detected.
- If there is no VFP instruction in the VFP11 Issue stage, the VFP11 coprocessor waits until one is issued. The next VFP instruction is the trigger instruction and is bounced.

When the MPCore processor returns from exception processing, it retries the trigger instruction.

19.4.3 Exception processing for CDP short vector instructions

For short vector instructions, any iteration might be exceptional. If an exceptional condition is detected for a vector iteration, the vector iterations issued before the exceptional iteration are allowed to complete and retire.

When a short vector iteration is found to be potentially exceptional, the following operations occur:

1. The EX flag, FPEXC[31], is set.
2. The source and destination register addresses are modified in the instruction word to point to the source and destination registers of the potentially exceptional iteration.
3. The FPINST register is loaded with the operation instruction word.
4. The VECITR field, FPEXC[10:8], is written with the number of iterations remaining after the potentially exceptional iteration.
5. The exceptional condition flags are set in the FPEXC.

19.4.4 Examples of exception detection for vector instructions

In Example 19-1, the FMULD instruction is a short vector operation with b011 in the LEN field for a length of four iterations and b00 in the STRIDE field for a vector stride of one. A potential Underflow exception is detected on the third iteration.

Example 19-1 Exceptional short vector FMULD followed by load/store instructions

```
FMULD D8, D12, D8   ; Short vector double-precision multiply of length 4
FLDD D0, [R5]       ; Load of 1 double-precision register
FSTMS R3, {S2-S9}   ; Store multiple of 8 single-precision registers
FLDS S8, [R9]       ; Load of 1 single-precision register
```

A double-precision multiply requires two cycles in the Execute 2 stage. The exception on the third iteration is detected in cycle 8. Before the FMULD exception is detected, the FLDD enters the Decode stage in cycle 2, and the FSTMS enters the Decode stage in cycle 3. The FLDD and the FSTMS complete execution and retire. The FLDS stalls in the Decode stage due to a resource conflict with the FSTMS and is the trigger instruction. It is bounced in cycle 9 and can be retried after exception processing. FPINST2 is invalid, and the FP2V flag, FPEXC[28], is not set.

Table 19-1 shows the pipeline stages for Example 19-1.

Table 19-1 Exceptional short vector FMULD followed by load/store instructions

	Instruction cycle number															
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
FMULD D8, D12, D8	D	I	E1	E2	E1	E2	E1	E2	-	-	-	-	-	-	-	-
FLDD D0, [R5]	-	D	I	E	M1	M2	W	-	-	-	-	-	-	-	-	-
FSTMS R3, {S2-S9}	-	-	D	I	E	M1	M2	W	W	W	W	-	-	-	-	-
FLDS S8, [R9]	-	-	-	D	D	D	D	I	*	-	-	-	-	-	-	-

After exception processing begins, the FPEXC register fields contain the following:

```
EX      1   The VFP11 coprocessor is in the exceptional state.
EN      1
FP2V    0   FPINST2 does not contain a valid instruction.
VECITR  000 One iteration remains after the exceptional iteration.
INV     0
UFC     1   Exception detected is a potential underflow.
OFC     0
```

```
IOC     0
```

The FPINST register contains the FMULD instruction with the following fields modified to reflect the register address of the third iteration.

```
Fd/D    1010/0 Destination of the third exceptional iteration is D10.
Fm/M    1010/0 Fm source of the third exceptional iteration is D10.
Fn/N    1110/0 Fn source of the third exceptional iteration is D14.
```

The FPINST2 register contains invalid data.

In Example 19-2, the first FADDS is a short vector operation with b001 in the LEN field for a vector length of two iterations and b00 in the STRIDE field for a vector stride of one. A potential Invalid Operation exception is detected in the second iteration. The second FADDS progresses to the Execute 1 stage and is captured in the FPINST2 register with the condition field changed to AL, the FP2V flag set, and is not the trigger instruction. The FMULS is the trigger instruction and bounces in cycle 6. It can be retried after exception processing.

Example 19-2 Exceptional short-vector FADDS with a FADDS in the pretrigger slot

```
FADDS S24, S26, S28 ; Vector single-precision add of length 2
FADDS S3, S4, S5 ; Scalar single-precision add
FMULS S12, S16, S16; Short vector single-precision multiply
```

Table 19-2 shows the pipeline stages for Example 19-2.

Table 19-2 Exceptional short vector FADDS with a FADDS in the pretrigger slot

	Instruction cycle number															
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
FADDS S24, S26, S28	D	I	E1	E1	E2	-	-	-	-	-	-	-	-	-	-	-
FADDS S3, S4, S5	-	D	D	I	E1	-	-	-	-	-	-	-	-	-	-	-
FMULS S12, S16, S16	-	-	-	D	I	*	-	-	-	-	-	-	-	-	-	-

After exception processing begins, the FPEXC register fields contains the following:

```
EX      1   The VFP11 coprocessor is in the exceptional state.
EN      1
FP2V    1   FPINST2 contains a valid instruction.
VECITR  111 No iterations remaining after exceptional iteration.
INV     0
```

```
UFC     0
OFC     0
IOC     1   Exception detected is a potential invalid operation.
```

The FPINST register contains the FADDS instruction with the following fields modified to reflect the register address of the second iteration:

```
Fd/D    1100/1  Destination is of the second exceptional iteration is S25.
Fn/N    1101/1  Fn source is of the second exceptional iteration is S27.
Fm/M    1110/1  Fm source is of the second exceptional iteration is S29.
```

The FPINST2 register contains the instruction word for the second FADDS with the condition field changed to AL.

In Example 19-3, FADDD is a short vector instruction with b011 in the LEN field for a vector length of four iterations and b00 in the STRIDE field for a vector stride of one. It has a potential Overflow exception in the first iteration, detected in cycle 4. The following FMACS is stalled in the Decode stage. The FMACS is the trigger instruction and can be retried after exception processing. FPINST2 is invalid and the FP2V flag is not set.

Example 19-3 Exceptional short vector FADDD with an FMACS trigger instruction

```
FADDD D4, D4, D12   ; Short vector double-precision add of length 4
FMACS S0, S3, S2    ; Scalar single-precision mac
```

Table 19-3 shows the pipeline stages for Example 19-3.

Table 19-3 Exceptional short vector FADDD with an FMACS trigger instruction

	Instruction cycle number															
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
FADDD D4, D4, D12	D	I	E1	E2	-	-	-	-	-	-	-	-	-	-	-	-
FMACS S0, S3, S2	-	D	D	I	*				-	-	-	-	-	-	-	-

After exception processing begins, the FPEXC register fields contain the following:

```
EX      1   The VFP11 coprocessor is in the exceptional state.
EN      1
FP2V    0   FPINST2 does not contain a valid instruction.
VECITR  010 Three iterations remain.
INV     0
UFC     0
```

```
OFC     1   Exception detected is a potential overflow.
IOC     0
```

The FPINST register contains the FADDD instruction with the following fields modified to reflect the register address of the first iteration:

```
Fd/D    0100/0  Destination of exceptional iteration is D4.
Fn/N    0100/0  Fn source of the first exceptional iteration is D4.
Fm/M    1100/0  Fm source of the first exceptional iteration is D12.
```

FPINST2 contains invalid data.

19.5 Input Subnormal exception

The IDC flag, FPSCR[7], is set whenever a floating-point operand is subnormal. The behavior of the VFP11 coprocessor with a subnormal input operand is a function of the FZ bit, FPSCR[24]. If FZ is not set, the VFP11 coprocessor bounces on the presence of a subnormal input. If FZ is set, the IDE bit, FPSCR[15], determines whether a bounce occurs.

19.5.1 Exception enabled

Setting the IDE bit enables Input Subnormal exceptions. An Input Subnormal exception sets the EX flag, FPEXC[31], the INV flag, FPEXC[7], and calls the Input Subnormal user trap handler. The source and destination registers for the instruction are unchanged in the VFP11 register file.

19.5.2 Exception disabled

Clearing the IDE bit disables Input Subnormal exceptions. In flush-to-zero mode, the result of the operation, with the subnormal input replaced with a positive zero, is completed and written to the register file. The IDC flag, FPSCR[7], is set.

19.6 Invalid Operation exception

An operation is *invalid* if the result cannot be represented, or if the result is not defined.

Table 19-4 shows the operand combinations that produce Invalid Operation exceptions. In addition to the conditions in Table 19-4, any CDP instruction other than FCPY, FNEG, or FABS causes an Invalid Operation exception if one or more of its operands is an SNaN (see Table 17-1 on page 17-5).

Table 19-4 Possible Invalid Operation exceptions

Instruction	Invalid Operation exceptions
FADD	(+infinity) + (–infinity) or (–infinity) + (+infinity).
FSUB	(+infinity) – (+infinity) or (–infinity) – (–infinity).
FCMPE/FCMPEZ	Any NaN operand
FMUL/FNMUL	Zero $\times$ ±infinity or ±infinity $\times$ zero.[a]
FDIV	Zero/zero or infinity/infinity.[a]
FMAC/FNMAC	Any condition that can cause an Invalid Operation exception for FMUL or FADD can cause an Invalid Operation exception for FMAC and FNMAC. The product generated by the FMAC or FNMAC multiply operation is considered in the detection of the Invalid Operation exception for the subsequent sum operation.
FMSC/FNMSC	Any of the conditions that can cause an Invalid Operation exception for FMUL or FSUB can cause an Invalid Operation exception for FMSC and FNMSC. The product generated by the FMSC or FNMSC multiply operation is considered in the detection of the Invalid Operation exception for the subsequent difference operation.
FSQRT	Source is less than 0.
FTOUI	Rounded result would lie outside the range $0 \leq \text{result} < 2^{32}$.
FTOSI	Rounded result would lie outside the range $-2^{31} \leq \text{result} < 2^{31}$.

a. In flush-to-zero mode, a subnormal input is treated as a positive zero for detecting an Invalid Operation exception.

19.6.1 Exception enabled

Setting the IOE bit, FPSCR[8], enables Invalid Operation exceptions.

The VFP11 coprocessor causes a bounce to support code for all the invalid operation conditions listed in Table 19-4 on page 19-15. Any arithmetic operation involving an SNaN also causes a bounce to support code. The VFP11 coprocessor detects most Invalid Operations exceptions conclusively but some are detected based on the possibility of an invalid operation. The potentially invalid operations are:

- FTOUI with a negative input. A small negative input might round to a zero, which is not an invalid condition.
- A float-to-integer conversion with a maximum exponent for the destination integer and any rounding mode other than round-towards-zero. The impact of rounding is unknown in the Execute 1 stage.
- An FMAC family operation with an infinity in the A operand and a potential product overflow when an infinity with the sign of the product would result in an invalid condition.

When the VFP11 coprocessor detects a potentially invalid condition, the EX flag, FPEXC[31], and the IOC flag, FPEXC[0], are set. The IOC flag in the FPSCR register, FPSCR[0], is not set by the hardware and must be set by the support code before calling the Invalid Operation user trap handler.

The support code determines the exception status of all bounced instructions. If an invalid condition exists, the Invalid Operation user trap handler is called. The source and destination registers for the instruction are valid in the VFP11 register file.

19.6.2 Exception disabled

If the IOE bit is not set, the VFP11 coprocessor writes a default NaN into the destination register for all operations except integer conversion operations.

Conversion of a floating-point value that is outside the range of the destination integer is an invalid condition rather than an overflow condition. When an invalid condition exists for a float-to-integer conversion, the VFP11 coprocessor delivers a default result to the destination register and sets the IOC flag, FPSCR[0]. Table 19-5 on page 19-17 shows the default results for input values after rounding.

If the VFP11 coprocessor is not in default NaN mode, an arithmetic instruction with an SNaN operand sets the IOC flag and causes a bounce to support code.

Note

A negative input to an unsigned conversion that does not round to a true zero in the conversion process sets the IOC flag, FPEXC[0].

Table 19-5 Default results for invalid conversion inputs

	FTOUIS and FTOUID		FTOSIS and FTOSID	
Input value after rounding	**Result**	**FPSCR IOC flag set?**	**Result**	**FPSCR IOC flag set?**
$x \geq 2^{32}$	`0xFFFFFFFF`	Yes	`0x7FFFFFFF`	Yes
$2^{31} \leq x < 2^{32}$	Integer	No	`0x7FFFFFFF`	Yes
$0 \leq x < 2^{31}$	Integer	No	Integer	No
$0 \geq x \geq -2^{31}$	`0x00000000`	Yes	Integer	No
$x < -2^{31}$	`0x00000000`	Yes	`0x80000000`	Yes
NaN	`0x00000000`	Yes	`0x00000000`	Yes
+infinity	`0xFFFFFFFF`	Yes	`0x7FFFFFFF`	Yes
–infinity	`0x00000000`	Yes	`0x80000000`	Yes

19.7 Division by Zero exception

The Division by Zero exception is generated for a division by zero of a normal or subnormal value. In flush-to-zero mode, a subnormal input is treated as a positive zero for detection of a division by zero. What happens depends on whether or not the Invalid Operation exception is enabled.

19.7.1 Exception enabled

If the DZE bit, FPSCR[9], is set, the Division by Zero user trap handler is called. The source and destination registers for the instruction are unchanged in the VFP11 register file.

19.7.2 Exception disabled

Clearing the DZE bit disables Division by Zero exceptions. A correctly signed infinity is written to the destination register, and the DZC flag, FPSCR[1], is set.

19.8 Overflow exception

When the OFE bit, FPSCR[10], is set, the hardware detects overflow pessimistically based on the preliminary calculation of the final exponent value. If the OFE bit is not set, the hardware detects overflow conclusively.

19.8.1 Exception enabled

Setting the OFE bit enables overflow exceptions. The VFP11 coprocessor detects most overflow conditions conclusively, but it detects some based on the possibility of overflow. The initial computation of the result exponent might be the maximum exponent or one less than the maximum exponent of the destination precision. Then the possibility of overflow due to significand overflow or rounding exists, but cannot be known in the first Execute stage. The VFP11 coprocessor bounces on such cases and uses the support code to determine the exceptional status of the operation. If there is no overflow, the support code writes the computed result to the destination register and does not set the OFC flag, FPSCR[2]. If there is an overflow, the intermediate result is written to the destination register, OFC is set, and the Overflow user trap handler is called. The support code sets or clears the IXC flag, FPSCR[4], as appropriate.

When the VFP11 coprocessor detects a potential overflow condition, the EX flag, FPEXC[31], and the OFC flag, FPEXC[2], are set. The OFC flag in the FPSCR register, FPSCR[2], is not set by the hardware and must be set by the support code before calling the user trap handler. The source and destination registers for the instruction are unchanged in the VFP11 register file. See *Arithmetic exceptions* on page 19-25 for the conditions that cause an overflow bounce.

19.8.2 Exception disabled

Clearing the OFE bit disables overflow exceptions. A correctly signed infinity or the largest signed finite number for the destination precision is written to the destination register as Table 19-6 shows. The OFC and IXC flags, FPSCR[2] and FPSCR[4], are set.

Table 19-6 Rounding mode overflow results

Rounding mode	Result
Round to nearest	Infinity, with the sign of the intermediate result.

Table 19-6 Rounding mode overflow results (continued)

Rounding mode	Result
Round towards zero	Largest magnitude value for the destination size, with the sign of the intermediate result.
Round towards plus infinity	Positive infinity if positive overflow. Largest negative value for the destination size if negative overflow.
Round towards minus infinity	Largest positive value for the destination size if positive overflow. Negative infinity if negative overflow.

19.9 Underflow exception

Underflow is detected pessimistically in non-RunFast mode. If the potential underflow is confirmed by the support code for an operation with a floating-point result, an underflow exception is generated. How this is confirmed depends on whether the VFP11 coprocessor is in flush-to-zero mode.

If the FZ bit is set, all underflowing results are forced to a positive signed zero and written to the destination register. The UFC flag is set in the FPSCR. No trap is taken. If the Underflow exception enable bit is set, it is ignored.

If the FZ bit is not set what happens next depends on whether the Underflow exception is enabled.

19.9.1 Exception enabled

Setting the UFE bit, FPSCR[11], enables Underflow exceptions. The VFP11 coprocessor detects most underflow conditions conclusively, but it detects some based on the possibility of an underflow. The initial computation of the result exponent might be below a threshold for the destination precision. In this case, the possibility of underflow due to massive cancellation exists, but cannot be known in the first Execute stage. The VFP11 coprocessor bounces on such cases and uses the support code to determine the exceptional status of the operation. Underflow is confirmed if the result of the operation after rounding is less in magnitude than the smallest normalized number in the destination format. If there is no underflow, either catastrophic or to a subnormal result, the support code writes the computed result to the destination register and returns without setting the UFC flag, FPSCR[3]. If there is underflow, regardless of any accuracy loss, the intermediate result is written to the destination register, UFC is set, and the Underflow user trap handler is called. The support code sets or clears the IXC flag, FPSCR[4], as appropriate.

When the VFP11 coprocessor detects a potential underflow condition, the EX flag, FPEXC[31], and the UFC flag, FPEXC[3], are set. The UFC flag in the FPSCR register is not set by the hardware and must be set by the support code before calling the user trap handler. The source and destination registers for the instruction are valid in the VFP11 register file. See section *Arithmetic exceptions* on page 19-25 for the conditions that cause an underflow bounce.

19.9.2 Exception disabled

Clearing the UFE bit, FPSCR[11], disables Underflow exceptions. When the FZ bit, FPSCR[24], is not set, the VFP11 coprocessor bounces on potential underflow cases in the same fashion as described in *Exception enabled*. The correct result is written to the destination register, setting the appropriate exception flags.

When the FZ bit is set, the VFP11 coprocessor makes the determination of underflow before rounding and flushes any result that underflows. A result that underflows returns a positive zero to the destination register and sets the UFC flag, FPSCR[3].

Note

The determination of an underflow condition in flush-to-zero mode is made before rounding rather than after. This means that the VFP11 coprocessor might not return the minimum normal value when rounding would have produced it. Instead, it flushes to zero an intermediate value with the minimum exponent for the destination precision, a fraction of all ones, and a round increment. If the intermediate value was the minimum normal value before the underflow condition test is made, it is not flushed to zero.

19.10 Inexact exception

The result of an arithmetic operation on two floating-point values can have more significant bits than the destination register can contain. When this happens, the result is rounded to a value that the destination register can hold and is said to be *inexact*.

The Inexact exception occurs whenever:

- a result is not equal to the computed result before rounding
- an untrapped Overflow exception occurs
- an untrapped Underflow exception occurs, and there is loss of accuracy.

Note

The Inexact exception occurs frequently in normal floating-point calculations and does not indicate a significant numerical error except in some specialized applications. Enabling the Inexact exception by setting the IXE bit, FPSCR[12], can significantly reduce the performance of the VFP11 coprocessor.

The VFP11 coprocessor handles the Inexact exception differently from the other floating-point exceptions. It has no mechanism for reporting inexact results to the software, but can handle the exception without software intervention as long as the IXE bit, FPSCR[12], is cleared, disabling Inexact exceptions.

19.10.1 Exception enabled

If the IXE bit, FPSCR[12], is set, all CDP instructions are bounced to the support code without any attempt to perform the calculation. The support code is then responsible for performing the calculation, determining if any exceptions have taken place, and handling them appropriately. If the support code detects an Inexact exception, it calls the Inexact user trap handler.

Note

- The inexact exception takes priority over all other exceptions.
- The inexact exception is taken precisely, unlike other exceptions. This means that when a CDP is bounced, because it is potentially imprecise, the instruction can be found at the address pointed to by r14-4 and is not stored in the FPINST register. There is never a pre-trigger instruction in the FPINST2 register.

19.10.2 Exception disabled

If the IXE bit, FPSCR[12], is not set, the VFP11 coprocessor writes the result to the destination register and sets the IXC flag, FPSCR[4].

19.11 Input exceptions

The VFP11 hardware processes most input operands without support code assistance. However, the hardware is incapable of processing some operands and bounces to support code to process the instruction. An arithmetic operation bounces with an Input exception when it has either of the following:

- a NaN operand or operands, and default NaN mode is not enabled
- a subnormal operand or operands, and flush-to-zero mode is not enabled.

Note

In default NaN mode, an SNaN input to an arithmetic operation causes an Invalid Operation exception. When the IOE bit, FPSCR[8], is set, the instruction bounces to the Invalid Operation user trap handler. When the IOE bit is clear, and the VFP11 coprocessor is not in default NaN mode, the instruction bounces to the support code.

19.12 Arithmetic exceptions

This section describes the conditions under which the VFP11 coprocessor bounces an arithmetic instruction based on the potential for the exception. It is the task of the support code to determine the actual exception status of the instruction. The support code must return either the result and appropriate exception status bits, or the intermediate result and a call to a user trap handler.

The following sections describe the circumstances in which arithmetic exceptions occur:

- *FADD and FSUB* on page 19-26
- *FCMP, FCMPZ, FCMPE, and FCMPEZ* on page 19-28
- *FMUL and FNMUL* on page 19-28
- *FMAC, FMSC, FNMAC, and FNMSC* on page 19-29
- *FDIV* on page 19-29
- *FSQRT* on page 19-31
- *FCPY, FABS, and FNEG* on page 19-31
- *FCVTDS and FCVTSD* on page 19-31
- *FUITO and FSITO* on page 19-31
- *FTOUI, FTOUIZ, FTOSI, and FTOSIZ* on page 19-32.

19.12.1 FADD and FSUB

In an addition or subtraction, the exponent is initially the larger of the two input exponents. For clarity, we define the operation as a *Like-Signed Addition* (LSA) or an *Unlike-Signed Addition* (USA). Table 19-7 specifies how this distinction is made. In the table, + indicates a positive operand, and – indicates a negative operand.

Table 19-7 LSA and USA determination

Instruction	Operand A sign	Operand B sign	Operation type
FADD	+	+	LSA
FADD	+	–	USA
FADD	–	+	USA
FADD	–	–	LSA
FSUB	+	+	USA
FSUB	+	–	LSA
FSUB	–	+	LSA
FSUB	–	–	USA

Because it is possible for an LSA operation to cause the exponent to be incremented if the significand overflows, overflow bounce ranges for an LSA are more pessimistic than they are for a USA. The LSA ranges are made slightly more pessimistic to incorporate FMAC instructions (see *FMAC, FMSC, FNMAC, and FNMSC* on page 19-29).

Underflow bounce ranges for a USA are more pessimistic than they are for an LSA. This is to accommodate a massive cancellation in which the result exponent is smaller than the larger operand exponent by as much as the length of the significand. The overflow range for a USA is slightly pessimistic (it is set to the LSA overflow range) to reduce the number of logic terms. Table 19-8 on page 19-27 shows the USA and LSA values and conditions. The exponent values in Table 19-8 on page 19-27 are in biased format.

Table 19-8 FADD family bounce thresholds

Initial result exponent value			Condition when not in flush-to-zero mode	
DP[a]	**SP**[b]	**Float value**	**SP**	**DP**
>0x7FF	-	DP overflow	-	Bounce
0x7FF	-	DP overflow, NaN, or infinity	-	Bounce
0x7FE	-	DP overflow	-	Bounce
0x7FD	-	DP overflow	-	Bounce
0x7FC	-	DP normal	-	Normal
>0x47F	>0xFF	SP overflow	Bounce	Normal
0x47F	0xFF	SP NaN or infinity	Bounce	Normal
0x47E	0xFE	SP overflow	Bounce	Normal
0x47D	0xFD	SP overflow	Bounce	Normal
0x47C	0xFC	SP normal	Normal	Normal
0x3FF	0x7F	e = 0 bias value	Normal	Normal
0x3A0	0x20	SP normal (LSA)	Minimum (USA)	Normal
0x39F	0x1F	SP underflow (USA)	Bounce (USA) or normal (LSA)	Normal
0x381	0x01	SP normal (LSA)	MIN (LSA)	Normal
0x380	0x00	SP subnormal	Bounce	Normal
<0x380	<0x00	SP underflow	Bounce	Normal
0x040	-	DP normal (USA)	-	Normal (LSA) or minimum (USA)
0x03F	-	DP underflow (USA)	-	Normal (LSA) or bounce (USA)
0x001	-	DP normal (LSA)	-	Minimum (LSA) or bounce (USA)
0x000	-	DP subnormal	-	Bounce
<0x000	-	DP underflow	-	Bounce

a. DP = double-precision.

b. SP = single-precision.

19.12.2 FCMP, FCMPZ, FCMPE, and FCMPEZ

Compare operations do not generate potential exceptions.

19.12.3 FMUL and FNMUL

Detection of a potential exception is based on the initial product exponent, which is the sum of the multiplicand and multiplier exponents. Table 19-9 shows the result for specific values of the initial product exponent. The exponent values in Table 19-9 are in biased format. The exponent can be incremented by a significand overflow condition, which is the cause for the additional bounce values near the real overflow threshold. The one additional value in the bounce range makes the FMUL and FNMUL overflow detection ranges identical to those in Table 19-8 on page 19-27.

Table 19-9 FMUL family bounce thresholds

Initial product exponent value			Condition in full-compliance mode	
DP[a]	**SP**[b]	**Float value**	**SP**	**DP**
`>0x7FF`	-	DP overflow	-	Bounce
`0x7FF`	-	DP NaN or infinity	-	Bounce
`0x7FE`	-	DP maximum normal	-	Bounce
`0x7FD`	-	DP normal	-	Bounce
`0x7FC`	-	DP normal	-	Normal
`>0x47F`	`>0xFF`	SP overflow	Bounce	Normal
`0x47F`	`0xFF`	SP NaN or infinity	Bounce	Normal
`0x47E`	`0xFE`	SP maximum normal	Bounce	Normal
`0x47D`	`0xFD`	SP normal	Bounce	Normal
`0x47C`	`0xFC`	SP normal	Normal	Normal
`0x3FF`	`0x7F`	e = 0 bias value	Normal	Normal
`0x381`	`0x01`	SP normal	Normal	Normal
`0x380`	`0x00`	SP subnormal	Bounce	Normal

Table 19-9 FMUL family bounce thresholds (continued)

Initial product exponent value			Condition in full-compliance mode	
DP[a]	**SP**[b]	**Float value**	**SP**	**DP**
`<0x380`	`<0x00`	SP underflow	Bounce	Normal
`0x001`	-	DP normal	-	Normal
`0x000`	-	DP subnormal	-	Bounce
`<0x000`	-	DP underflow	-	Bounce

a. DP = double-precision.
b. SP = single-precision.

19.12.4 FMAC, FMSC, FNMAC, and FNMSC

The FMAC family of operations adds to the potential overflow range by generating significand values from zero up to but not including four. In this case it is possible for the final exponent to require incrementing by two to normalize the significand.

The bounce thresholds for the FADD family in Table 19-8 on page 19-27 and for the FMUL family in Table 19-9 on page 19-28 incorporate this additional factor. Those ranges are used to detect potential exceptions for the FMAC family.

19.12.5 FDIV

The thresholds for divide are simple and based only on the difference of the exponents of the dividend and the divisor. It is not possible in a divide operation for the significand to overflow and cause an increment of the exponent. However, it is possible for the significand to require a single bit left shift and the exponent to be decremented for normalization. To reduce logic complexity, the overflow ranges are the same as those of the LSA operations in *FADD and FSUB* on page 19-26. The underflow ranges include the minimum normal exponent, `0x01` for single-precision and `0x001` for double-precision. Table 19-10 shows the FDIV bounce thresholds. The exponent values shown in Table 19-10 are in biased format.

Table 19-10 FDIV bounce thresholds

Initial quotient exponent value			**Condition in full-compliance mode**	
DP[a]	**SP**[b]	**Float value**	**SP**	**DP**
`>0x7FF`	-	DP overflow	-	Bounce
`0x7FF`	-	DP NaN or infinity	-	Bounce
`0x7FE`	-	DP maximum normal	-	Bounce
`0x7FD`	-	DP normal	-	Bounce
`0x7FC`	-	DP normal	-	Normal
`>0x47F`	`>0xFF`	SP overflow	Bounce	Normal
`0x47F`	`0xFF`	SP NaN or infinity	Bounce	Normal
`0x47E`	`0xFE`	SP maximum normal	Bounce	Normal
`0x47D`	`0xFD`	SP normal	Bounce	Normal
`0x47C`	`0xFC`	SP normal	Normal	Normal
`0x3FF`	`0x7F`	e = 0 bias value	Normal	Normal
`0x382`	`0x02`	SP normal	Normal	Normal
`0x381`	`0x01`	SP normal	Bounce	Normal
`0x380`	`0x00`	SP subnormal	Bounce	Normal
`<0x380`	`<0x00`	SP underflow	Bounce	Normal
`0x002`	-	DP normal	-	Normal
`0x001`	-	DP normal	-	Bounce
`0x000`	-	DP subnormal	-	Bounce
`<0x000`	-	DP underflow	-	Bounce

a. DP = double-precision.
b. SP = single-precision.

19.12.6 FSQRT

It is not possible for FSQRT to overflow or underflow.

19.12.7 FCPY, FABS, and FNEG

It is not possible for FCPY, FABS, or FNEG to bounce for any operand.

19.12.8 FCVTDS and FCVTSD

Only the FCVTSD operation is capable of overflow or underflow. To reduce logic complexity, the overflow ranges are the same as the LSA ranges. Table 19-11 shows the FCVTSD bounce conditions. The exponent values shown in Table 19-11 are in biased format.

Table 19-11 FCVTSD bounce thresholds

Double-precision operand exponent value	Float value	FCVTSD condition in full-compliance mode
`>0x47F`	SP[a] overflow	Bounce
`0x47F`	SP NaN or infinity	Bounce
`0x47E`	SP maximum normal	Bounce
`0x47D`	SP normal	Bounce
`0x47C`	SP normal	Normal
`0x3FF`	e = 0 bias value	Normal
`0x381`	SP normal	Normal
`0x380`	SP subnormal	Bounce
`<0x380`	SP underflow	Bounce

a. SP = single-precision.

19.12.9 FUITO and FSITO

It is not possible to generate overflow or underflow in an integer-to-float conversion.

19.12.10 FTOUI, FTOUIZ, FTOSI, and FTOSIZ

Float-to-integer conversions generate Invalid Operation exceptions rather than Overflow or Underflow exceptions. To support signed conversions with round-towards-zero rounding in the maximum range possible for C, C++, and Java compiled code, the thresholds for pessimistic bouncing are different for the various rounding modes.

Table 19-12 on page 19-33 and Table 19-13 on page 19-34 use the following notation:

In the *VFP Response* column, the response notations are:

all	These input values are bounced for all rounding modes.
S	These input values are bounced for signed conversions in all rounding modes.
SnZ	These input values are bounced for signed conversions in all rounding modes except round-towards-zero.
U	These input values are bounced for unsigned conversions in all rounding modes.
UnZ	These input values are bounced for unsigned conversions in all rounding modes except round-towards-zero.

In the *Unsigned results and Signed results* columns, the rounding mode notations are:

N	Round-to-nearest mode.
P	Round-towards-plus-infinity mode.
M	Round-towards-minus infinity mode.
Z	Round-towards-zero mode.

Table 19-12 on page 19-33 shows the single-precision float-to-integer bounce range and the results returned for exceptional conditions. The exponent values shown in Table 19-12 on page 19-33 are in biased format.

Table 19-12 Single-precision float-to-integer bounce thresholds and stored results

Floating-point value	Integer value	Unsigned result	Status	Signed result	Status	VFP11 response
NaN	-	`0x00000000`	Invalid	`0x00000000`	Invalid	Bounce all
`0x7F800000`	+infinity	`0xFFFFFFFF`	Invalid	`0x7FFFFFFF`	Invalid	Bounce all
`0x7F7FFFFF` to `0x4F800000`	+maximum SP[a] to 2^{32}	`0xFFFFFFFF`	Invalid	`0x7FFFFFFF`	Invalid	Bounce all
`0x4F7FFFFF` to `0x4F000000`	$2^{32} – 2^8$ to 2^{31}	`0xFFFFFF00` to `0x80000000`	Valid	`0x7FFFFFFF`	Invalid	Bounce S UnZ
`0x4EFFFFFF` to `0x4E800000`	$2^{31} – 2^7$ to 2^{30}	`0x7FFFFF80` to `0x40000000`	Valid	`0x7FFFFF80` to `0x40000000`	Valid	Bounce SnZ
`0x4E7FFFFF` to `0x00000000`	$2^{30} – 2^6$ to +0	`0x3FFFFFC0` to `0x00000000`	Valid	`0x3FFFFFC0` to `0x00000000`	Valid	No bounce
`0x80000000` to `0xCE7FFFFF`	–0 to $–2^{30} + 2^6$	`0x00000000`	Invalid[b]	`0x00000000` to `0xC0000040`	Valid	Bounce U
`0xCE800000` to `0xCEFFFFFF`	$–2^{30}$ to $–2^{31} + 2^7$)	`0x00000000`	Invalid	`0xC0000000` to `0x80000080`	Valid	Bounce U
`0xCF000000`	$–2^{31}$	`0x00000000`	Invalid	`0x80000000`	Valid	Bounce U SnZ
`0xCF000000` to `0xFF7FFFFF`	$–2^{31}$ to –maximum SP	`0x00000000`	Invalid	`0x80000000`	Invalid	Bounce all
`0xFF800000`	–infinity	`0x00000000`	Invalid	`0x80000000`	Invalid	Bounce all

a. SP = single-precision.
b. A negative input value that rounds to a zero result returns zero and is not invalid.

Table 19-13 shows the double-precision float-to-integer bounce range and the results returned for exceptional conditions.

Table 19-13 Double-precision float-to-integer bounce thresholds and stored results

Floating-point value	Integer value	Unsigned result	Status	Signed result	Status	VFP11 response
NaN	-	`0x00000000`	Invalid	`0x00000000`	Invalid	Bounce all
`0x7FF00000 00000000`	+infinity	`0xFFFFFFFF`	Invalid	`0x7FFFFFFF`	Invalid	Bounce all
`0x7FEFFFFF FFFFFFFF`	+maximum DP[a]	`0xFFFFFFFF`	Invalid	`0x7FFFFFFF`	Invalid	
to	to					Bounce all
`0x41F00000 00000000`	2^{32}					
`0x41EFFFFF FFFFFFFF`	$2^{32} - 2^{21}$	`0xFFFFFFFF` N, P	Invalid			
to	to	`0xFFFFFFFF` Z, M	Valid	`0x7FFFFFFF`	Invalid	Bounce S UnZ
`0x41EFFFFF FFF00000`	$2^{32} - 2^{-1}$					
`0x41EFFFFF FFEFFFFF`	$2^{32} - 2^{-1} - 2^{21}$	`0xFFFFFFFF` P	Invalid			
to	to	`0xFFFFFFFF` N, Z, M	Valid	`0x7FFFFFFF`	Invalid	Bounce S UnZ
`0x41EFFFFF FFE00001`	$2^{32} - 2^{0} + 2^{-21}$					
`0x41EFFFFF FFE00000`	$2^{32} - 2^{0}$	`0xFFFFFFFF`	Valid	`0x7FFFFFFF`	Invalid	Bounce S UnZ
to	to	to				
`0x41E00000 00000000`	2^{31}	`0x80000000`				
`0x41DFFFFF FFFFFFFF`	$2^{31} - 2^{22}$	`0x80000000` N, P	Valid	`0x7FFFFFFF` N, P	Invalid	
to	to	`0x7FFFFFFF` Z, M	Valid	`0x7FFFFFFF` Z, M	Valid	Bounce SnZ
`0x41DFFFFF FFE00000`	$2^{31} - 2^{-1}$					
`0x41DFFFFF FFDFFFFF`	$2^{31} - 2^{-1} - 2^{-22}$	`0x80000000` P	Valid	`0x7FFFFFFF` P	Invalid	Bounce SnZ
to	to	`0x7FFFFFFF` N, Z, M	Valid	`0x7FFFFFFF` N, Z, M	Valid	
`0x41DFFFFF FFC00001`	$2^{31} - 2^{0} + 2^{-22}$					
`0x41DFFFFF FFC00000`	$2^{31} - 2^{0}$	`0x7FFFFFFF`	Valid	`0x7FFFFFFF`	Valid	Bounce SnZ
to	to	to	Valid	to		
`0x41D00000 00000000`	2^{30}	`0x40000000`		`0x40000000`	Valid	
`0x41CFFFFF FFFFFFFF`	$2^{30} - 2^{23}$	`0x40000000` N, P	Valid	`0x40000000` N, P	Valid	
to	to	`0x3FFFFFFF` Z, M		`0x3FFFFFFF` Z, M	Valid	Bounce none
`0x00000000 00000000`	+0	to	Valid	to		
		`0x00000000`		`0x00000000`	Valid	

Table 19-13 Double-precision float-to-integer bounce thresholds and stored results (continued)

Floating-point value	Integer value	Unsigned result	Status	Signed result	Status	VFP11 response
`0x80000000 00000000` to `0xC1CFFFFF FFFFFFFF`	–0 to $-2^{30} + 2^{-23}$	`0x00000000`[b]	Invalid	`0x00000000` to `0xC0000001` Z, P; `0xC0000000` N, M	Valid; Valid; Valid	Bounce U
`0xC1D00000 00000000` to `0xC1DFFFFF FFFFFFFF`	-2^{30} to $-2^{31} + 2^{-22}$	`0x00000000`	Invalid	`0xC0000000` to `0x80000001` Z, P; `0x80000000` N, M	Valid; Valid; Valid	Bounce U
`0xC1E00000 00000000`	-2^{31}	`0x00000000`	Invalid	`0x80000000`	Valid	Bounce U SnZ
`0xC1E00000 00000001` to `0xC1E00000 00100000`	$-2^{31} - 2^{-21}$ to $-2^{31} - 2^{-1}$	`0x00000000`	Invalid	`0x80000000` N, Z, P; `0x80000000` M	Valid; Invalid	Bounce U SnZ
`0xC1E00000 00100001` to `0xC1E00000 001FFFFF`	$-2^{31} - 2^{-1} - 2^{-21}$ to $2^{31} - 2^{0} + 2^{-21}$	`0x00000000`	Invalid	`0x80000000` Z, P; `0x80000000` N, M	Valid; Invalid	Bounce U SnZ
`0xC1E00000 00200000` to `0xFFEFFFFF FFFFFFFF`	$2^{31} - 2^{0}$ to –maximum DP	`0x00000000`	Invalid	`0x80000000`	Invalid	Bounce all
`0xFFF00000 00000000`	–infinity	`0x00000000`	Invalid	`0x00000000`	Invalid	Bounce all

a. DP = double-precision.

b. A negative input value that rounds to a zero result returns zero and is not invalid.

Part C
Appendices

Appendix A
Signal Descriptions

This appendix lists and describes the MPCore signals. It contains the following sections:

Note

Table A-1 on page A-2 to Table A-15 on page A-13 show output signals. These are set to 0 on reset unless otherwise stated.

A.1 AXI interface signals

All the signals described below relate to the MPCore level of hierarchy. See the *AMBA AXI Protocol v1.0 Specification* for more information.

A.1.1 Master port 0

Table A-1 shows the master port 0 read address channel signals.

Table A-1 Master port 0 read address channel

Signal	Input / Output	Description
ARREADY0	Input	Address Ready
ARVALID0	Output	Address Valid
ARADDR0[31:0]	Output	Address

ARLEN0[3:0]	Output	Burst Length
		Burst length that gives the exact number of transfer
		b0000 = 1 data transfer
		b0001 = 2 data transfers
		b0010 = 3 data transfers
		b0011 = 4 data transfers

ARSIZE0[2:0]	Output	Burst size
		b000 = 8-bit transfer
		b001 = 16-bit transfer
		b010 = 32-bit transfer
		b011 = 64-bit transfer

ARBURST0[1:0]	Output	Burst type
		b01 = INCR incrementing burst
		b10 = WRAP Wrapping burst

ARLOCK0[1:0]	Output	Lock type
		b00 = normal access
		b01 = exclusive access
		b10 = locked access

ARCACHE0[3:0]	Output	Cache type giving additional information about cachable characteristics

Table A-1 Master port 0 read address channel (continued)

Signal	Input / Output	Description
ARPROT0[2:0]	Output	Protection Type
ARID0[3:0]	Output	Address ID
ARUSER0[4:0]	Output	ARUSER0[4:1] Inner memory region attributes. ARUSER0[0] Memory region Shared attribute.

Table A-2 shows the master port 0 read channel signals.

Table A-2 Master port 0 read channel

Signal	Input / Output	Description
RVALID0	Input	Read Valid
RLAST0	Input	Read Last
RDATA0[63:0]	Input	Read Data
RRESP0[1:0]	Input	Read Response
RID0[3:0]	Input	Read ID
RREADY0	Output	Read Ready

Table A-3 shows the master port 0 write address channel signals.

Table A-3 Master port 0 write address channel

Signal	Input / Output	Description
AWREADY0	Input	Address Ready
AWVALID0	Output	Address Valid
AWADDR0[31:0]	Output	Address
AWLEN0[3:0]	Output	Burst Length
AWSIZE0[2:0]	Output	Burst Size
AWBURST0[1:0]	Output	Burst Type

Table A-3 Master port 0 write address channel (continued)

Signal	Input / Output	Description

AWLOCK0[1:0]	Output	Lock Type
		b00 = normal access
		b01 = exclusive access

AWCACHE0[3:0]	Output	Cache Type
AWPROT0[2:0]	Output	Protection Type
AWID0[3:0]	Output	Address ID
AWUSER0[6:0]	Output	AWUSER0[6] clean eviction transfer AWUSER0[5] eviction transfer AWUSER0[4:1] Inner memory region attributes AWUSER0[0] memory region Shared attribute

Table A-4 shows the master port 0 write channel signals.

Table A-4 Master port 0 write channel

Signal	Input / Output	Description
WREADY0	Input	Write Ready
WVALID0	Output	Write Valid
WLAST0	Output	Write Last
WDATA0[63:0]	Output	Write Data
WSTRB0[7:0]	Output	Write Strobes
WID0[3:0]	Output	Write ID

Table A-5 shows the master port 0 write response channel signals.

Table A-5 Master port 0 write response channel

Signal	Input / Output	Description
BVALID0	Input	Response Valid

Table A-5 Master port 0 write response channel

Signal	Input / Output	Description
BRESP0[1:0]	Input	Write Response
BID0[3:0]	Input	Response ID
BREADY0	Output	Response Ready

A.1.2 Master port 1

Table A-6 shows the master port 1 read address channel signals.

Table A-6 Master port 1 read address channel

Signal	Input / Output	Description
ARREADY1	Input	Address Ready
ARVALID1	Output	Address Valid
ARADDR1[31:0]	Output	Address
ARLEN1[3:0]	Output	Burst Length
ARSIZE1[2:0]	Output	Burst Size
ARBURST1[1:0]	Output	Burst Type

ARLOCK1[1:0]	Output	Lock Type
		b00 = normal access
		b01 = exclusive access
		b10 = locked access

ARCACHE1[3:0]	Output	Cache Type
ARPROT1[2:0]	Output	Protection Type
ARID1[3:0]	Output	Address ID
ARUSER1[4:0]	Output	ARUSER1[4:1] Inner memory region attributes ARUSER1[0] memory region Shared attribute

Table A-7 shows the master port 1 read channel signals.

Table A-7 Master port 1 read channel

Signal	Input / Output	Description
RVALID1	Input	Read Valid
RLAST1	Input	Read Last
RDATA1[63:0]	Input	Read Data
RRESP1[1:0]	Input	Read Response
RID1[3:0]	Input	Read ID
RREADY1	Output	Read Ready

Table A-8 shows the master port 1 write address channel signals.

Table A-8 Master port 1 write address channel

Signal	Input / Output	Description
AWREADY1	Input	Address Ready
AWVALID1	Output	Address Valid
AWADDR1[31:0]	Output	Address
AWLEN1[3:0]	Output	Burst Length
AWSIZE1[2:0]	Output	Burst Size
AWBURST1[1:0]	Output	Burst Type
AWLOCK1[1:0]	Output	Lock Type
AWCACHE1[3:0]	Output	Cache Type
AWPROT1[2:0]	Output	Protection Type
AWID1[3:0]	Output	Address ID
AWUSER1[6:0]	Output	AWUSER1[6] clean eviction transfer AWUSER1[5] eviction transfer AWUSER1[4:1] Inner memory region attributes AWUSER1[0] memory region Shared attribute

Table A-9 shows the master port 1 write channel signals.

Table A-9 Master port 1 write channel

Signal	Input / Output	Description
WREADY1	Input	Write Ready
WVALID1	Output	Write Valid
WLAST1	Output	Write Last
WDATA1[63:0]	Output	Write Data
WSTRB1[7:0]	Output	Write Strobes
WID1[3:0]	Output	Write ID

Table A-10 shows the master port 1 write response channel signals.

Table A-10 Master port 1 write response channel

Signal	Input / Output	Description
BVALID1	Input	Response Valid
BRESP1[1:0]	Input	Write Response
BID1[3:0]	Input	Response ID
BREADY1	Output	Response Ready

A.2 Interrupt lines

Table A-11 shows the interrupt line signals.

Table A-11 Interrupt line signals

Signal	Input / Output	Description
INT[n:0][a]	Input	Interrupt distributor interrupt lines. n can be 31, 63,…, up to 223 by increments of 32. If there are no interrupt lines this pin is removed.
nIRQ[3:0]	Input	CPU legacy IRQ request input lines
nFIQ[3:0]	Input	CPU private FIQ request input lines

a. The minimum pulse width of signals driving external interrupt lines is two CPU clock cycles.

A.3 Debug interface

Table A-12 shows the debug interface signals.

Table A-12 Debug interface signals

Signal	Input / Output	Description
TCK	Input	Test Clock
nTRST	Input	Debug reset (active low)
TDI	Input	Debug TDI
TMS	Input	Debug TMS
EDBGRQ[3:0]	Input	External Debug Request
DBGEN	Input	Debug Enable
CPUTDI[3:0]	Input	Nonsynchronized CPU DBGTDI input
TDO[3:0]	Output	Debug TDO
DBGNOPWRDWN[3:0]	Output	Debugger has requested MP11 CPU is not powered down
DBGNTDOEN[3:0]	Output	Debug TDO enable
COMMTX[3:0]	Output	Comms Channels Transmit
COMMRX[3:0]	Output	Comms Channels Receive
DBGACK[3:0]	Output	Debug Acknowledge
RTCK	Output	Return test Clock
DBGTDI	Output	Synchronized version of TDI

A.4 MBIST interface

Table A-13 shows the MBIST interface signals.

Table A-13 MBIST interface signals

Signal	Input / Output	Description
MBISTADDR[10:0]	Input	MBIST Address Bus
MBISTCE[18:0]	Input	MBIST Chip Enable
MBISTDIN[63:0]	Input	MBIST Data In
MBISTnRESET	Input	MBIST reset
MBISTWE	Input	MBIST Write Enable
MTESTON	Input	MBIST Test is Enabled
MBISTDOUT[255:0]	Output	MBIST Data Out

See Appendix C *MBIST Controller and Dispatch Unit* for a description of MBIST.

A.5 Power control interface

Table A-14 shows power control interface signals.

Table A-14 Power control interface signals

Signal	Input / Output	Description
PWRCTLO0[1:0]	Output	b0x CPU0 must be powered on b10 CPU0 can enter dormant mode b11 CPU0 can enter powered-off mode.
PWRCTLO1[1:0]	Output	b0x CPU1 must be powered on b10 CPU1 can enter dormant mode b11 CPU1 can enter powered-off mode.
PWRCTLO2[1:0]	Output	b0x CPU2 must be powered on b10 CPU2 can enter dormant mode b11 CPU2 can enter powered-off mode.
PWRCTLO3[1:0]	Output	b0x CPU3 must be powered on b10 CPU3 can enter dormant mode b11 CPU3 can enter powered-off mode
PWRCTLI0[1:0]	Input	Reset value for CPU status register [1:0]
PWRCTLI1[1:0]	Input	Reset value for CPU status register [3:2]
PWRCTLI2[1:0]	Input	Reset value for CPU status register [5:4]
PWRCTLI3[1:0]	Input	Reset value for CPU status register [7:6]
RAMCLAMP[4:0]	Input	RAM Clamps control signals **RAMCLAMP[4]** SCU RAMs **RAMCLAMP[3]** CPU3 RAMs **RAMCLAMP[2]** CPU2 RAMs **RAMCLAMP[1]** CPU1 RAMs **RAMCLAMP[0]** CPU0 RAMs

Table A-14 Power control interface signals

Signal	Input / Output	Description
CPUCLAMP[3:0]	Input	CPU interrupt interface clamps control signals **CPUCLAMP[3]** CPU3 I/F **CPUCLAMP[2]** CPU2 I/F **CPUCLAMP[1]** CPU1 I/F **CPUCLAMP[0]** CPU0 I/F
BISTCLAMP	Input	BIST interface clamp control signal
DEBUGCLAMP	Input	Debug interface clamp control signal

A.6 Miscellaneous signals

Table A-15 shows miscellaneous signals.

Table A-15 Miscellaneous signals

Signal	Input/Output	Description
CLK	Input	Global clock.
ACLKEN	Input	Master AXI buses clock enable.
nWDRESET[3:0]	Input	Individual watchdog resets.
nCPURESET[3:0]	Input	Individual MP11 CPU resets.
nSCURESET	Input	SCU global reset.
nPORESET[3:0]	Input	MP11 CPU debug logic reset.
CFGEND[1:0]	Input	Endianness configuration.
VINITHI	Input	When HIGH, indicates high vecs mode.
FIQISNMI[3:0]	Input	Disable FIQ mask bit in CPSR per MP11 CPU so that FIQ acts as NMI.
CLUSTERID[3:0]	Input	Value read in CPU ID register field, bits[11:8].
PERIPHBASE[18:0]	Input	Specifies base address for private peripherals memory mapping.
MASTER1EN	Input	Number of master ports selected 0 = Master 0 only is used 1 = Master 0 and 1 are used.
PMUIRQ[11:0]	Output	Interrupt requests by system metrics Bit 0 for CPU0 Bit 1 for CPU1 Bit 2 for CPU2 Bit 3 for CPU3 Bits[11-4] for the SCU.
SMPnAMP[3:0]	Output	Signals AMP or SMP mode for each MP11 CPU.
RESETREQ[3:0]	Output	Individual watchdog reset requests.
STANDBYWFI[3:0]	Output	Indicates if an MP11 CPU is in WFI state.

Table A-15 Miscellaneous signals (continued)

Signal	Input/Output	Description
STANDBYWFE[3:0]	Output	Indicates if an MP11 CPU is in WFE state.
EVNTIN	Input	Event Input for MP11 CPUs wake-up from WFE state.
EVNTOUT	Output	Event Output (active when one SEV instruction is executed).

A.7 Scan test signals

Table A-16 lists the scan test signals.

Table A-16 Scan test signals

Name	Direction	Description
SCANMODE	Input	In scan test mode
SE1	Input	Scan enable
SE2	Input	Scan enable
SE3	Input	Scan enable
SE4	Input	Scan enable
SE5	Input	Scan enable

Appendix B
AC Characteristics

This chapter gives the timing diagram and timing parameters for the ARM11 MPCore processor. It contains the following sections:

- *MPCore timing* on page B-2
- *MPCore signal timing parameters* on page B-3.

B.1 MPCore timing

The AMBA bus interface of the ARM11 MPCore processor conforms to the *AMBA Specification*. See the *AMBA Specification* for the relevant timing diagrams for the ARM11 MPCore processor.

B.2 MPCore signal timing parameters

Signal timing parameters are given in:

- *Registered signals*
- *Unregistered signals*.

B.2.1 Registered signals

To ensure ease of integration of the ARM11 MPCore processor into embedded applications, and to simplify synthesis flow, the following design techniques have been used:

- a single rising edge clock times all activity
- all signals and buses are unidirectional
- all inputs are required to be synchronous to the relevant clock, CLK, or HCLK.

These techniques simplify the definition of the ARM11 MPCore processor top-level signals because all outputs change from the rising edge and all inputs are sampled with the rising edge of the clock. In addition, all signals are either input or output only. Bidirectional signals are not used.

B.2.2 Unregistered signals

The unregistered input signals are:

- **ARREADY0**, **ARREADY1**
- **RVALID0**, **RVALID1**
- **AWREADY0**, **AWREADY1**
- **WREADY0**, **WREADY1**
- **BVALID0**, **BVALID1**
- **INT[n:0]**, **nIRQ[3:0]**
- **CLK**, **ACLKEN**,**nWDRESET[3:0]**, **nCPURESET[3:0]**, **nSCURESET**, and **nPORESET[3:0]**.

There are no unregistered output signals.

Figure B-1 on page B-4 shows the target timing parameters for unregistered signals. The timing parameter T is the internal clock latency of the clock buffer tree, and it is dependent on process technology and design parameters. Timing parameters ending with suffix h represent hold times. Timing parameters ending with suffix d represent delay times. Contact your silicon supplier for more details.

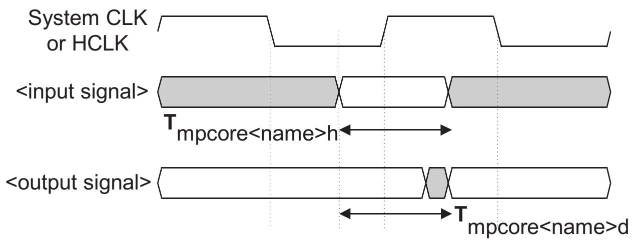


Figure B-1 Target timing parameters for unregistered signals

Note

Actual clock frequencies and input and output timing constraints vary according to application requirements and the silicon process technologies used. The maximum operating clock frequencies attained by ARM devices increases over time as a result.

Appendix C
MBIST Controller and Dispatch Unit

This chapter describes the MBIST Controller and MBIST Dispatch Unit. It contains the following sections:

C.1 About MBIST

Memory Built-in Self Test (MBIST) provides a way to directly test the compiled RAM memory cells used in the MPCore cores, MP11 CPU0-MP11 CPU3, and the Snoop Control Unit. MBIST writes and reads all locations of the RAM to ensure that the cells are operating correctly. MPCore MBIST can access up to four cores in parallel, provided they use the same RAM size configuration.

MBIST mode take priority over all other modes. RAM arrays are only accessible to the MPCore MBIST block when MBIST mode is activated with the **MTESTON** pin. In functional mode, **MTESTON** must be kept LOW.

Figure C-1 shows the functional blocks of the MPCore MBIST module.

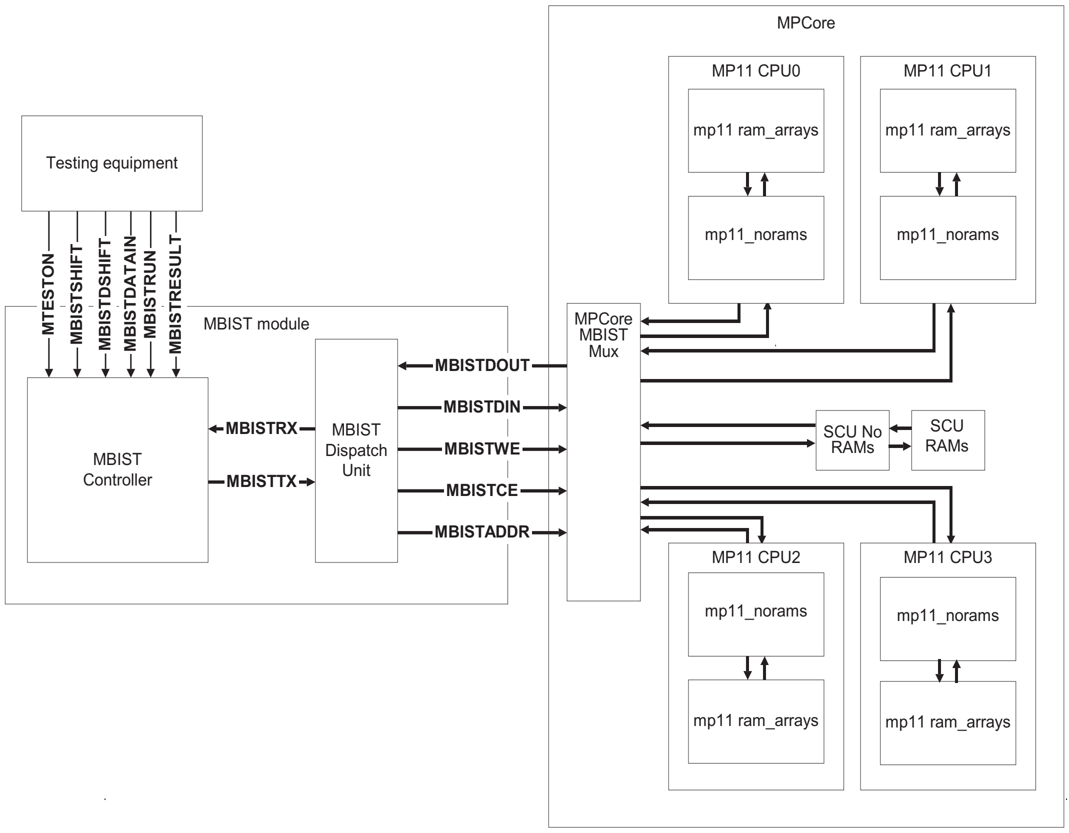


Figure C-1 MBIST block diagram

C.2 MBIST Controller and MBIST Dispatch Unit

The MBIST module consists of:

- an MBIST Controller
- an MBIST Dispatch Unit.

The controller and dispatch unit can test all RAM arrays in the ARM11 MPCore processor. This port connects to the RAM through existing multiplexers within the core, removing the need to increase the length of the critical path.

Two signals, **MBISTTX[11:0]** and **MBISTRX[5:0]**, are used to exchange data between the MBIST controller and MBIST dispatch unit.

C.2.1 MBIST Instruction Register

MBIST executes loaded instructions stored in the MBIST Instruction Register. The Instruction Register is 56 bits wide. It is split between the control and dispatch unit:

- 18 bits are used for the control unit
- 38 bits are used for the dispatch unit.

The MBIST Instruction Register is loaded through the serial port using the **MBISTDATAIN** control unit when **MBISTSHIFT** is asserted.

MBISTDATAIN is serially passed through the MBIST Controller to the MBIST Dispatch Unit using the **MBISTTX[3**] port, when **MBISTSHIFT** is asserted.

C.3 MBIST Controller

The MBIST Controller decodes the instruction to be performed that has been shifted serially into a 53-bit shift register. A finite state machine then sequences through the operations required to perform the required algorithm.

C.4 MBIST Dispatch Unit

The MBIST Dispatch Unit uses the signals supplied by the MBIST Controller to perform memory accesses using the MBIST Interface. The MBIST Dispatch Unit also evaluates data obtained as the result of a read operation against expected data. This evaluation requires the use of registers to store the pipelining of memory accesses within the core. The dispatch unit also generates the addresses required at each stage of the test.

C.4.1 Address scrambler

For many tests neighboring cells must be accessed in turn. This is not the same as accesses to sequential addresses. It is therefore necessary to scramble the target address to perform a physical to logical mapping, so that sequential logical addresses access neighboring cells. The address scrambler is located in the MBIST Dispatch Unit. Partners must modify the address scrambler to perform the mapping required for their RAMs.

C.5 MBIST signal descriptions

This section describes the MBIST signals:

- *MBIST Tester and MBIST Controller signals*
- *Controller and Dispatch Unit signals* on page C-7
- *MBIST Dispatch Unit and MPCore signals* on page C-9.

C.5.1 MBIST Tester and MBIST Controller signals

Table C-1shows the MBIST Tester and MBIST Controller signals.

Table C-1 MBIST Tester and MBIST Controller signals

Name	Direction relative to MBIST controller	Description
MTESTON	Input	Switches multiplexors to give access to the RAMs. Must be HIGH during Memory BIST mode.
MBISTSHIFT	Input	Enables serial loading of the MBIST Instruction Register through **MBISTDATAIN**. This signal must be HIGH during instruction register loading, and LOW otherwise. For more information about instruction register format see *Shift Register and Fail Datalog Format* on page C-11.
MBISTDSHIFT	Input	Enables serial output of failing data log using **MBISTRESULT[5:2]**. When detecting a fail you must set **MBISTDSHIFT** to output a data log. The MBIST Controller stalls until MBISTDSHIFT becomes LOW. For more information about data log format see *Shift Register and Fail Datalog Format* on page C-11.
MBISTDATAIN	Input	Serial input port used to load the MBIST Instruction Register when **MBISTSHIFT** is HIGH.
MBISTRUN	Input	Enables execution of the MBIST instruction previously loaded. It must be HIGH during MBIST testing and it must be LOW during instruction register loading.
MBISTRESULT[5:0]	Output	MBIST results and status.

MBISTRESULT values

Table C-2 shows the MBISTRESULT signals.

Table C-2 MBISTRESULT signal descriptions

Name	Direction	Description
MBISTRESULT[5]	Output	Serially outputs a data log for cpu3 when **MBISTDSHIFT** is asserted. Serially outputs the previous contents of the MBIST Instruction Register if **MBISTSHIFT** is asserted. Outputs an address expired signal otherwise.
MBISTRESULT[4]	Output	Serially outputs a data log for cpu2 when **MBISTDSHIFT** is asserted. Serially outputs the previous contents of the MBIST Instruction Register if **MBISTSHIFT** is asserted. Outputs an address expired signal otherwise.
MBISTRESULT[3]	Output	Serially outputs a data log for cpu1 when **MBISTDSHIFT** is asserted. Serially outputs the previous contents of the MBIST Instruction Register if **MBISTSHIFT** is asserted. Outputs an address expired signal otherwise.
MBISTRESULT[2]	Output	Serially outputs a data log for cpu0 when **MBISTDSHIFT** is asserted. Serially outputs the previous contents of the MBIST Instruction Register if **MBISTSHIFT** is asserted. Outputs an address expired signal otherwise.
MBISTRESULT[1]	Output	Outputs a fail detection signal if compare fails on any of the MP11 CPUs.
MBISTRESULT[0]	Output	Outputs a test finished signal.

C.5.2 Controller and Dispatch Unit signals

Signals from the MBIST Controller to the MBIST Dispatch Unit are combined onto the MBISTTX bus. Table C-3 shows the MBISTTX bus bit assignments.

Table C-3 MBISTTX bus bit assignments

Bit	Equivalent Signal
[11]	Stall Dispatch Unit pipeline.
[10]	Update data counter in Dispatch Unit and force it to use a predefined data seed for go/nogo test.
[9]	BitmapMode requested.
[8]	Yfast performed if 1, Xfast else.
[7]	Direction bit for updating address. Up if 0, down if 1.
[6]	Instruct Dispatch Unit that MBIST is in read mode.

Table C-3 MBISTTX bus bit assignments (continued)

Bit	Equivalent Signal
[5]	Instruct Dispatch Unit that MBIST is in write mode. Registered to create **MBISTWE**. No compare is performed in write mode.
[4]	DataCkbd.
[3]	Invert data seed bits in run mode or enable Dispatch Unit instruction load if **MBISTSHIFT** asserted.
[2]	AddrSacrifice.
[1]	Instruct Dispatch Unit to modify address in accordance with **MBISTTX[8:7]**.
[0]	Reset Address to start address.

Signals from the MBIST Dispatch Unit to the MBIST Controller are grouped on the MBISTRX bus.

When instruction shift is enabled, data in the BIST Control Instruction Register is shifted to dispatch on bit 3 (AddrExpire in normal operation). Table C-4 shows the MBISTRX bit assignments.

Table C-4 MBISTR[5:0]

Bit	Equivalent Signal
[5]	AddrExpire / InstOut cpu3
[4]	AddrExpire / InstOut cpu2
[3]	AddrExpire / InstOut cpu1
[2]	AddrExpire / InstOut cpu0
[1]	Stall
[0]	Fail bit (one fault on at least one MP11 CPU)

C.5.3 MBIST Dispatch Unit and MPCore signals

Table C-5 shows the MBIST Dispatch Unit signals.

Table C-5 MBIST Dispatch Unit signals

Name	Direction	Description
MTESTON	Input	Switches multiplexors to give access to the RAMs. Must be HIGH during Memory BIST mode.
MBISTDIN[63:0]	Input	Data to the RAMs. Not all RAMs use the full width.
MBISTADDR[10:0]	Input	Address. Not all RAMs use the full address width.
MBISTCE[17:0]	Input	Chip enables for each of the RAMs.
MBISTWE	Input	Global write enable going to all of the RAMs.
MBISTDOUT[255:0]	Output	Data out for all of the RAMs. The RAM is selected using **MBISTCE**.

To enable testing of all CPUs of MPCore in parallel, **MBISTDOUT** is 256 bits wide. Table C-6 shows the mappings of the MP11 CPUs to the **MBISTDOUT** bits.

Table C-6 CPU mappings to MBISTOUT bits

CPU name	CPU Data bits	Corresponding MBISTDOUT Bits
CPU0	[63:0]	[63:0]
CPU1	[63:0]	[127:64]
CPU2	[63:0]	[191:128]
CPU3	[63:0]	[255:192]

Table C-7 shows **MBISTCE[17:0]** encoding.

Table C-7 MBISTCE encodings

MBISTCE bit	RAM
MBISTCE[17]	SCU tag RAM way 3 and SCU tag RAM way 2
MBISTCE[16]	SCU tag RAM way 1 and SCU tag RAM way 0
MBISTCE[15]	DData RAM way 3
MBISTCE[14]	DData RAM way 2

Table C-7 MBISTCE encodings

MBISTCE bit	RAM
MBISTCE[13]	DData RAM way 1
MBISTCE[12]	DData RAM way 0
MBISTCE[11]	DTagRAM way 3 and DTagRAM way 2
MBISTCE[10]	DTagRAM way 1 and DTagRAM way 0
MBISTCE[9]	DDirty RAM
MBISTCE[8]	IDataRAM array 7 and IDataRAM array 6
MBISTCE[7]	IDataRAM array 5 and IDataRAM array 4
MBISTCE[6]	IDataRAM array 3 and IDataRAM array 2
MBISTCE[5]	IDataRAM array 1 and IDataRAM array 0
MBISTCE[4]	ITagRAM way 3 and ITagRAM way 2
MBISTCE[3]	ITagRAM way 1 and ITagRAM way 0
MBISTCE[2]	BTAC RAMs
MBISTCE[1]	TLB RAM array 1
MBISTCE[0]	TLB RAM array 0

C.6 Shift Register and Fail Datalog Format

The Shift Instruction Register is a 56-bit wide serially loaded register. It is split between the MBIST Controller, 18 bits, and the MBIST Dispatch Unit, 38 bits.

The MBIST Instruction Register is loaded serially from bit 0 to bit 57 using **MBISTDATAIN**. Table C-8 shows the bit assignments.

Table C-8 MBIST Instruction Register bit assignments

Bits	Description
[55:50]	Pattern selection. See Table C-25 on page C-25.
[49:44]	Control signal (bitmap, stop on fail)
[43:41]	Write latency
[40:38]	Read latency

Table C-9 shows the MBIST Dispatch Unit bit assignments.

Table C-9 MBIST Dispatch Unit bit assignments

Bits	Description
[37:33]	One hot encoded CPU enable. This bus, CPUON [3:0], enables comparison to be done on requested cpus. For comparison on MP11 CPU3, set cpuon[3]. For comparison on MP11 CPU2, set cpuon[2]. For comparison on MP11 CPU1, set cpuon[1]. For comparison on MP11 CPU0, set cpuon[0].
[33:30]	Max X address. Number of columns.
[29:26]	Max Y address. Number of rows.
[25:22]	Data Seed. Can be overridden by controller for go/nogo test.

Table C-9 MBIST Dispatch Unit bit assignments (continued)

Bits	Description
[21:4]	One hot encoded array enable. See encoding below.
[3:2]	Column width from 4 to 32 00 column width is 4. 01 column width is 8. 10 column width is 16. 11 column width is 32.
[1:0]	Cache size 00 cache size is 16KB. 01 cache size is 16KB. 10 cache size is 32KB. 11 cache size is 64KB.

C.7 Fail data log

The fail data log is 89 bits wide for each CPU. Table C-10 shows the bit assignments.

Table C-10 Data log bit assignments

Bits	Description
[88:79]	One hot encoded array enable. See **MBISTCE[17:0]** for more details
[79:68]	Failing address
[67:4]	Data read compare from MP11 CPUs, failing bits are set to 1
[3:0]	Four bits data seed used during test

C.8 Testing RAM

Table C-11 shows the configuration options in `mpcore_mbist_defs.v`. Your chosen options must match the MPCore RTL.

Table C-11 RTL options

Variable name	Purpose	Default
`BIST_NB_CPU`	Indicates the number of MP11 CPUs implemented in MPCore	4
`BIST_CPU0_PRESENT`	CPU0 is present	Present
`BIST_CPU1_PRESENT`	CPU1 is present	Present
`BIST_CPU2_PRESENT`	CPU2 is present	Present
`BIST_CPU3_PRESENT`	CPU3 is present	Present

For each CPU of MPCore, and the corresponding tag RAMs in the SCU, each of the RAMs can be accessed using the interface in the following way, assuming maximum RAM sizes. Table C-12 shows the correspondences between the RAMs and the MBISTCE bits.

Table C-12 RAM accesses using MBISTCE

RAM Name	MBISTCE bit	Data Bits	Max Address Bits
SCUTagRAM way 3	[17]	[21:0]	[8:0]
SCUTagRAM way 2	[17]	[43:22]	[8:0]
SCUTagRAM way 1	[16]	[21:0]	[8:0]
SCUTagRAM way 0	[16]	[43:22]	[8:0]
DDataRAM3	[15]	[63:0]	[10:0]
DDataRAM2	[14]	[63:0]	[10:0]
DDataRAM1	[13]	[63:0]	[10:0]
DDataRAM0	[12]	[63:0]	[10:0]
DTagRAM2	[11]	[21:0]	[8:0]
DTagRAM3	[11]	[43:22]	[8:0]
DTagRAM0	[10]	[21:0]	[8:0]

Table C-12 RAM accesses using MBISTCE (continued)

RAM Name	MBISTCE bit	Data Bits	Max Address Bits
DTagRAM1	[10]	[43:22]	[8:0]
DDirtyRAM	[9]	[23:0]	[8:0]
IDataRAM7	[8]	[63:32]	[10:0]
IDataRAM6	[8]	[31:0]	[10:0]
IDataRAM5	[7]	[63:32]	[10:0]
IDataRAM4	[7]	[31:0]	[10:0]
IDataRAM3	[6]	[63:32]	[10:0]
IDataRAM2	[6]	[31:0]	[10:0]
IDataRAM1	[5]	[63:32]	[10:0]
IDataRAM0	[5]	[31:0]	[10:0]
ITagRAM way 3	[4]	[20:0]	[8:0]
ITagRAM way 2	[4]	[41:21]	[8:0]
ITagRAM way 1	[3]	[20:0]	[8:0]
ITagRAM way 0	[3]	[41:21]	[8:0]
PUBtacDataRAM	[2]	[31:0]	[6:0]
PUBtacTagRAM	[2]	[63:32]	[6:0]
TLBRAM1	[1]	[59:0]	[4:0]
TLBRAM0	[0]	[59:0]	[4:0]

C.9 Testing MP11 CPU RAMs

Testing MP11 CPU RAMs is described in:

- *Testing MP11 Dside data RAM*
- *Testing MP11 Dside tag RAM* on page C-17
- *Testing MP11 Iside data RAM* on page C-18
- *Testing MP11 Iside tag RAM* on page C-19
- *Testing MP11 data dirty RAM* on page C-20
- *Testing MP11 TLB RAM* on page C-21
- *Testing MP11 BTAC RAM* on page C-22.

C.9.1 Testing MP11 Dside data RAM

Dside data RAM is made up of four ways and four 32-bit high blocks per MP11 CPU. Ways are checked separately using **MBISTCE[15:12]** as the enable signals. Table C-13 shows the MBIST signals and the ways tested.

Table C-13 MBIST signals and ways

MBIST signal	Way	Data banks
MBISTCE[15]	Way0	u_data_bank0 and u_data_bank4
MBISTCE[14]	Way1	u_data_bank1 and u_data_bank5
MBISTCE[13]	Way2	u_data_bank2 and u_data_bank6
MBISTCE[12]	Way3	u_data_bank3 and u_data_bank7

Table C-14 shows the RAM arrays used for each cache size.

Table C-14 Data cache size and Dside data RAM arrays

Data cache size	Number of blocks per MP11 CPU	Block size	Address bus
16KB	8	512x32	**MBISTADDR[8:0]**
32KB	8	1024x32	**MBISTADDR[9:0]**
64KB	8	2048x32	**MBISTADDR[10:0]**

Each Dside data array is 64 bits wide. So, data writes and data reads use the **MBISTDIN[63:0]** and **MBISTDOUT[63:0]** mapping for each MP11 CPU shown in Figure C-2. n has the values 0,1,2, or 3.

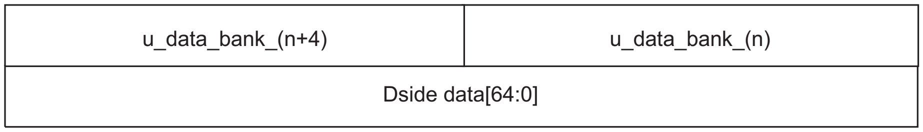


Figure C-2 Data mapping for MBIST

C.9.2 Testing MP11 Dside tag RAM

Dside data RAM is made of four ways and four 22-bit wide blocks. Ways are checked two by two using **MBISTCE[11:10]** as the enable signal.

Table C-15 MBIST signals and ways for Dside tag RAM

MBIST signal	Ways	Data banks
MBISTCE[11]	Way0 and Way1	u_tag_ram0 and u_tag_ram1
MBISTCE[10]	Way2 and Way3	u_tag_ram2 and u_tag_ram3

Table C-16 shows the RAM arrays used for each cache size.

Table C-16 Data cache size and tag RAM arrays

Data cache size	Number of blocks per MP11 CPU	Block size	Address bus
16KB	4	128x22	**MBISTADDR[6:0]**
32KB	4	256x22	**MBISTADDR[7:0]**
64KB	4	512x22	**MBISTADDR[8:0]**

Each Dside tag array is 22 bits wide. So, data writes and data reads use the following **MBISTDIN[63:0]** and **MBISTDOUT[63:0]** mapping for each MP11 CPU shown in Figure C-3 on page C-18.

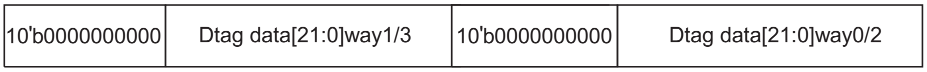


Figure C-3 Dside tag RAM mapping

C.9.3 Testing MP11 Iside data RAM

Iside data RAM is made of four 21-bit high blocks per MP11 CPU. Blocks are checked two by two using **MBISTCE[8:5]** as the enable signal.

Table C-17 MBIST enable signals and Iside data RAM bocks

MBIST signal	Iside data RAM blocks
MBISTCE[8]	u_idata_bank0 and u_idata_bank1
MBISTCE[7]	u_idata_bank1 and u_idata_bank5
MBISTCE[6]	u_idata_bank2 and u_idata_bank6
MBISTCE[5]	u_idata_bank3 and u_idata_bank7

Table C-18 shows the RAM arrays used for each cache size.

Table C-18 Iside cache size and data RAM arrays

Instruction cache size	Number of blocks per MP11 CPU	Block size	Address bus
16KB	8	512x32	**MBISTADDR[8:0]**
32KB	8	1024x32	**MBISTADDR[9:0]**
64KB	8	2048x32	**MBISTADDR[10:0]**

Each Iside data array is 64 bits wide. So, data writes and data reads use the following **MBISTDIN[63:0]** and **MBISTDOUT[63:0]** mapping for each MP11 CPU shown in Figure C-4 on page C-19. on page C-17 n has the value 0, 1, 2, or 3.

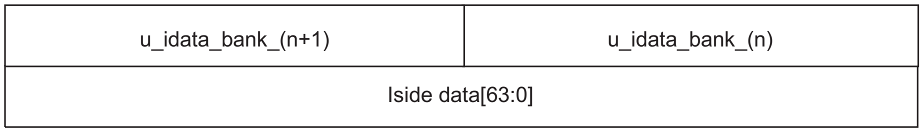


Figure C-4 Iside data array mapping

C.9.4 Testing MP11 Iside tag RAM

Iside tag RAM is made of four ways and four 2-bit wide blocks. Ways are checked two by two using **MBISTCE[4:3]** as the enable signal.

Table C-19 MBIST signals and ways for Iside tag RAM

MBIST signal	Ways	Data banks
MBISTCE[4]	Way0 and way1	u_tag_ram0 and u_tag_ram1
MBISTCE[3]	Way2 and way3	u_tag_ram2 and u_tag_ram3

Table C-20 shows the RAM arrays used for each cache size.

Table C-20 Cache sizes and iside tag RAM arrays

Instruction cache size	Number of blocks per MP11 CPU	Block size	Address bus
16KB	4	128x21	**MBISTADDR[6:0]**
32KB	4	256x21	**MBISTADDR[7:0]**
64KB	4	512x21	**MBISTADDR[8:0]**

Each Iside tag array is 21 bits wide. So data writes and data reads use the following **MBISTDIN[63:0]** and **MBISTDOUT[63:0]** mapping for each MP11 CPU shown in Figure C-5 on page C-20.

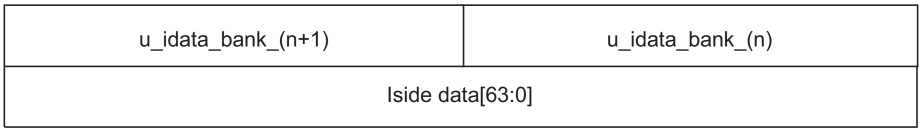


Figure C-5 Iside data RAM mapping

Depending on cache size some data bits are unused.

- For a 16KB instruction cache, comparison is performed on instruction tag data[20:0].
- For a 32KB instruction cache, comparison is performed on instruction tag data[20:1].
- For a 64KB instruction cache, comparison is performed on instruction tag data[20:2].

C.9.5 Testing MP11 data dirty RAM

Data dirty data ram is made of one 24-bit wide block enabled with MBISTCE[9]. Table C-21 shows the RAM arrays used for each cache size.

Table C-21 Cache sizes and Data dirty RAMs arrays

Cache size	Number of blocks per MP11 CPU	Block size	Address bus
16Kbytes	1	128x24	**MBISTADDR[6:0]**
32Kbytes	1	256x24	**MBISTADDR[7:0]**
64Kbytes	1	512x24	**MBISTADDR[8:0]**

Data dirty RAM is 24 bits wide. So, data writes and data reads use the **MBISTDIN[63:0]** and **MBISTDOUT[63:0]** mapping for each MP11 CPU shown in Figure C-6. on page C-17 n has the value 0, 1, 2, or 3.

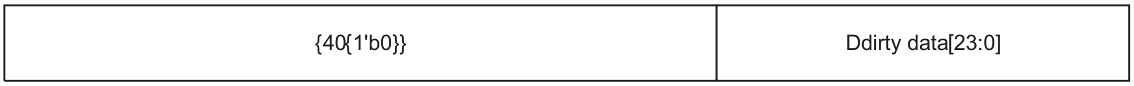


Figure C-6 Data dirty RAM mapping

C.9.6 Testing MP11 TLB RAM

Regardless of the instruction cache size or the data cache size, TLB RAMs are organized as two 32x60 RAM arrays enabled using **MBISTCE[1:0]** as the enable signal. Table C-22 shows the TLB RAm arrays and the MBIST enable signals.

Table C-22 TLB RAMs and MBIST signals

MBIST signal	TLB RAM
MBISTCE[1]	u_tlbram0
MBISTCE[0]	u_tlbram1

Figure C-7 shows the TLB RAM array organization in an MP11 four CPU configuration from the MBIST point of view.

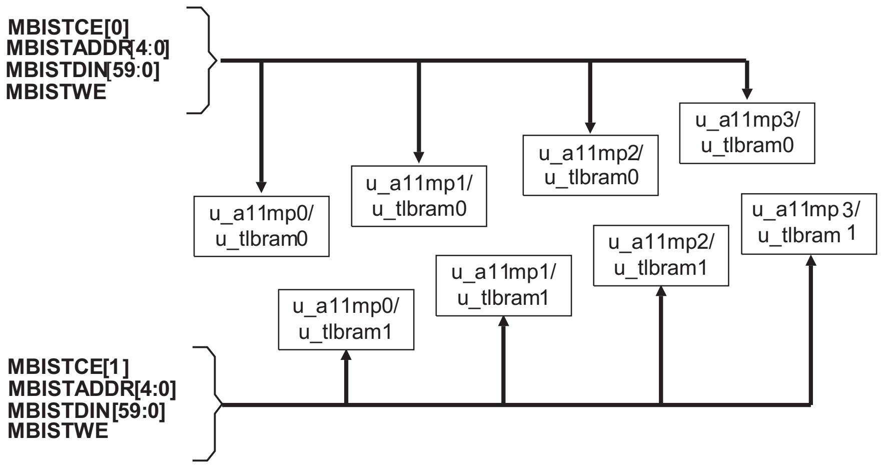


Figure C-7 TLB RAM organization with four MP11 CPUs

Each TLB RAM array is 60 bits wide. So, data writes and data reads use the following **MBISTDIN** and **MBISTDOUT** mapping for each MP11 CPU shown in Figure C-8.

4'b0000	TLB data[59:0]

Figure C-8 TLB mapping

C.9.7 Testing MP11 BTAC RAM

Branch target RAM is organized as one 128x63 RAM block enabled using **MBISTCE[2]** as shown in Figure C-9.

1'b0	BTAC data[62:0]

Figure C-9 BTAC mapping

C.10 Testing MP11 SCU RAM

SCU RAM arrays are enabled using **MBISTCE[17:16]**:

Table C-23 Enabling SCU RAM arrays

Signal	Ways	RAM arrays
MBISTCE[17]	Way 1 and Way 0	`u_tag_ram0`, `u_tag_ram1`, `u_tag_ram4`, `u_tag_ram5`, `u_tag_ram8`, `u_tag_ram9`, `u_tag_ram12`, and `u_tag_ram13`.
MBISTCE[16]	Way 2 and Way 3	`u_tag_ram2`, `u_tag_ram3`, `u_tag_ram6`, `u_tag_ram7`, `u_tag_ram10`, `u_tag_ram11`, `u_tag_ram14`, and `u_tag_ram15`.

Figure C-10 shows the SCU RAM array organization in a four MP11 CPU configuration from the MBIST point of view.

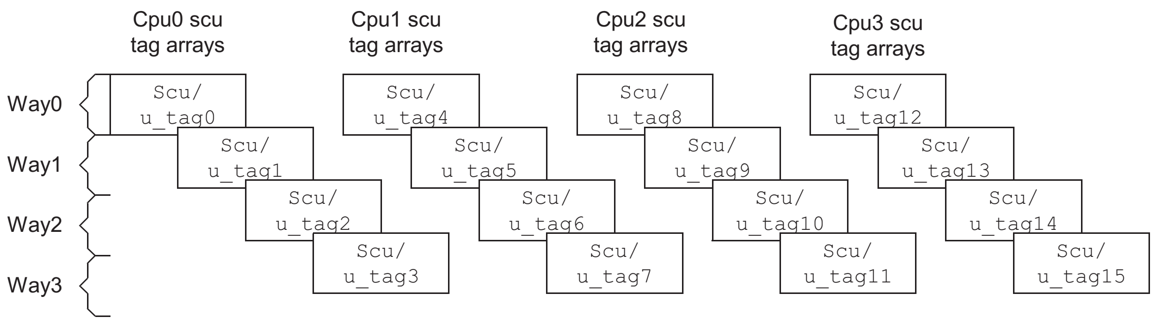


Figure C-10 SCU RAM array organization

Table C-24 shows the RAM arrays used for each Data Cache size.

Table C-24 SCU RAM arrays and Data cache sizes

Data Cache size	Number of blocks per MP11 CPU	Block size	Address bus
16KB	4	128x22	**MBISTADDR[6:0]**
32KB	4	256x22	**MBISTADDR[7:0]**
64KB	4	518x22	**MBISTADDR[8:0]**

Each SCU RAM is 22 bits wide. So data writes and data reads use the following **MBISTDIN[63]** and **MBISTDOUT[63:0]** mapping for each MP11 CPU.

C.11 Test patterns

Table C-25 shows the Instruction Register values and the MBIST test patterns

Table C-25 Instruction Register values and MBIST test patterns

Instruction Register [55:50]	Pattern name	Description
000000	WriteSolids	Write data (16 x data seed) at address location. Increment address X fast
000001	ReadSolids	Read data (16 x data seed) at address location. Increment address X fast.
000010	WriteCkbd	Write checkerboard data (16 x data seed) setting data polarity by using XOR (Xaddr[0],Yaddr[0])
000011	ReadCkbd	Read checkerboard data (16xdataseed) setting data polarity by using XOR(Xaddr[0],Yaddr[0]
000100	March C+	1. Write Solids pattern incrementing Xfast. 2. Read data, write inverted data, read inverted data at one location then increment address Xfast. 3. Reset Address to 0. 4. Read inverted data, write data, read data at one location then increment address Xfast. 5. Set address to address max. 6. Read data, write inverted data, read inverted data at one location then decrement address Xfast. 7. Set address to address max. 8. Read inverted data, write data, read data at one location then decrement address Xfast. 9. Set address to address max. 10. Read Solids pattern, decrementing address Xfast.
000101	PttnFail	Tests MBIST failure detection 1. Insert an error (invert data polarity for a write solids pattern) every 16 address to check the error detection logic detection, the first error at `0x0F`, then `0x1F` and so on. 2. Then for each location read data (so faulty each 16 index), write inverted data, read inverted data, increment address. 3. Then for each location read inverted data, write data, decrementing address. 4. Finish with a read solid incrementing address.

Table C-25 Instruction Register values and MBIST test patterns (continued)

Instruction Register [55:50]	Pattern name	Description
000110	RW_Xmarch	1. Write Solids pattern incrementing address Xfast. 2. Read data, write inverted data, read inverted data at one location then increment address Xfast. 3. Set address to address max. 4. Read inverted data, write data, read data at one location then decrement address Xfast. 5. Read Solids pattern.
000111	RW_Y march	Like RW_Xmarch but Yfast
001000	RWR_Xmarch	Increment decrement wordline fast march
001001	RWR_Ymarch	Increment decrement bitline fast march
001010	Bang	Description 1. Write solid pattern, incrementing address Xfast then reset address at 0 and read data at address location. 2. Write inverted data at address location. 3. Do 6 consecutive write inverted data at corresponding address in sacrificial row (row 0).
001011	MarchCy	March C yfast (column fast)
111111	Go / No Go	2*W/RCkbd, RWR Ymarch, X Bang 1. Write Checkerboard using 5 as dataseed incrementing address Xfast. 2. Reset Address to 0. 3. Read Checkerboard using 5 as dataseed incrementing address Xfast. 4. Reset Address to 0. 5. Write Checkerboard using 5 as dataseed incrementing address Xfast. 6. Reset Address to 0. 7. Read Checkerboard using 5 as dataseed incrementing address Xfast. 8. Perform a RWR Y march pattern. 9. Do a Bang pattern.

Appendix D
Scan chain ordering with RVI

This appendix describes RVI scan chain ordering. It contains the following section:

D.1 Scan chain ordering with RVI

RVI displays the scan chain order with the last device in the JTAG chain as zero. Table D-1 shows the MP11 CPUs and their physical JTAG chain positions with respect to RVI scan chain ordering.

Table D-1 RVI ordering, MP11 CPUID, and physical JTAG positions

RVI ordering	MP11 CPU ID	Physical JTAG chain position
3	0	First
2	1	Second
1	2	Third
0	3	Fourth

For example, if your scan chain has an ARM7 in the chain as the last unit it appears as shown in Table D-2.

Table D-2 One additional item in the scan chain

RVI ordering	MP11 CPU ID	Physical JTAG chain position
4	0	First
3	1	Second
2	2	Third
1	3	Fourth
0	ARM7	Fifth

However, if your scan chain has an ARM7 in the chain as the first unit and an ARM9 as the last unit they appear as shown in Table D-3.

Table D-3 Two additional items in the scan chain

RVI ordering	MP11 CPU ID	Physical JTAG chain position
5	ARM7	First
4	0	Second
3	1	Third

Table D-3 Two additional items in the scan chain (continued)

RVI ordering	MP11 CPU ID	Physical JTAG chain position
2	2	Fourth
1	3	Fifth
0	ARM9	Sixth

See the *RVI User Guide* for more information.

Appendix E
IEM

This appendix describes the provision of IEM in ARM11 MPCore processors. It contains the following sections:

Note

The ARM11 MPCore processor is IEM enabled but the level of support for the technology depends on the specific implementation.

E.1 Purpose of IEM

The purpose of IEM technology is to provide a dynamic optimization between processor performance and power consumption

E.1.1 Structure of IEM

The ARM11 MPCore processor provides a number of features that enable the processor voltage to vary relative to the voltage of the rest of the system. For this purpose the processor optionally implements placeholders for level shifters and clamps for some inputs and outputs including the debug interface and MBIST signals.

In addition, AXI IEM register slices are provided alongside the ARM11 MPCore processor IP.

Figure E-1 on page E-3 shows the structure for IEM in the processor. See the *Configuration Manual* for more information.

Note

You are free to implement wrappers on the remaining signals depending on your SoC architecture.

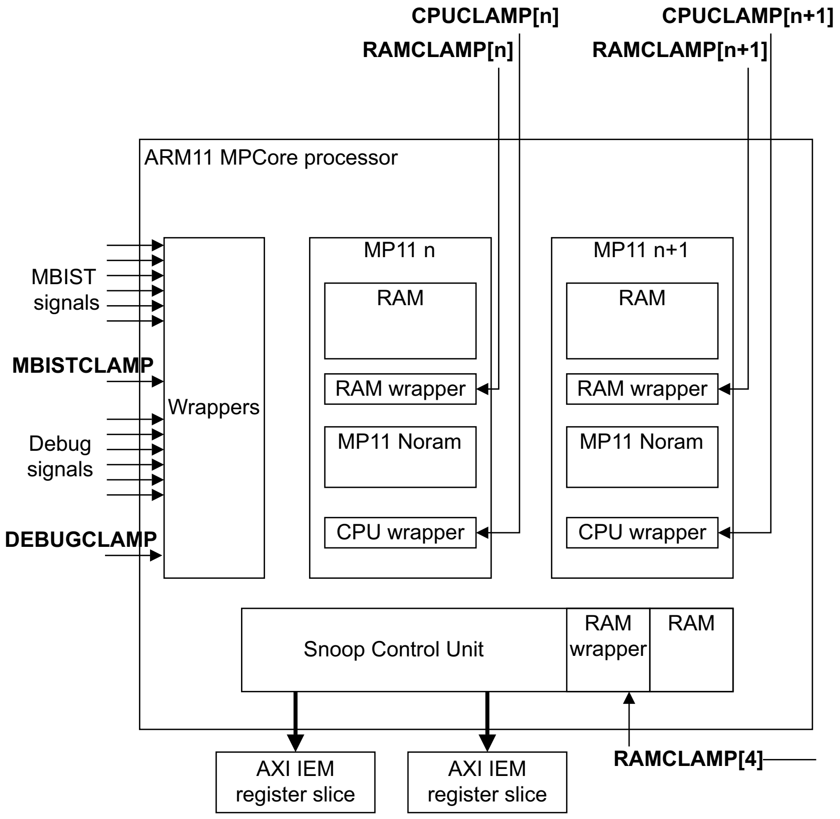


Figure E-1 IEM structure

E.1.2 Operation of IEM

IEM balances performance and power consumption by dynamic alteration of the processor clock frequency and supply voltage. **CLAMPEN** is provided to control the clamp cells between VCore and VSoc. Figure E-1 shows this.

E.1.3 Use of IEM

To use IEM the processor must be implemented with appropriate register slices and included in a SoC that contains an *Intelligent Energy Controller* (IEC™). For example systems, see the *Intelligent Energy Controller Technical Overview*.

IEM is functionally transparent to the user.

E.2 About AXI register slices

This section describes the external register slices and FIFOs used to provide an asynchronous AXI interface. The top level structure of IEM interface is made of:

- a register slice in the VCore domain, made of FIFOs of various depths
- a level shifter wrapper made of upshifter and downshifter cells.

  For ease of implementation this wrapper is split into two wrappers, one instantiating only downshifter cells, the other only upshifter cells.
- a register slice in the VSoc domain, made of FIFOs of various depths.

Figure E-2 shows the AXI register slices and level shifters in the ARM11 MPCore processor. Channel direction is given with respect to main data flow for each channel. Some signals are propagated backward to ensure correct handshaking.

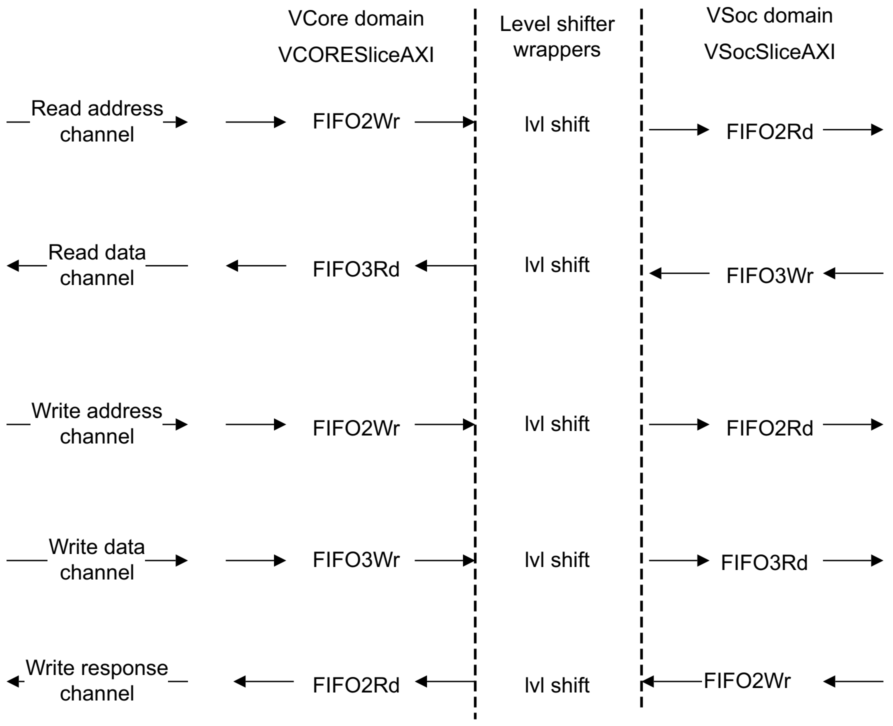


Figure E-2 AXI register slices and level shifters

These register slices can be dynamically bypassed when the IEC requests high performance. When the core is running at maximum performance, the master and slave clocks to the IEM slice can be multiplexed out. This removes all the latency that the synchronizers introduce. Figure E-3 shows the IEC request/acknowledge interface.

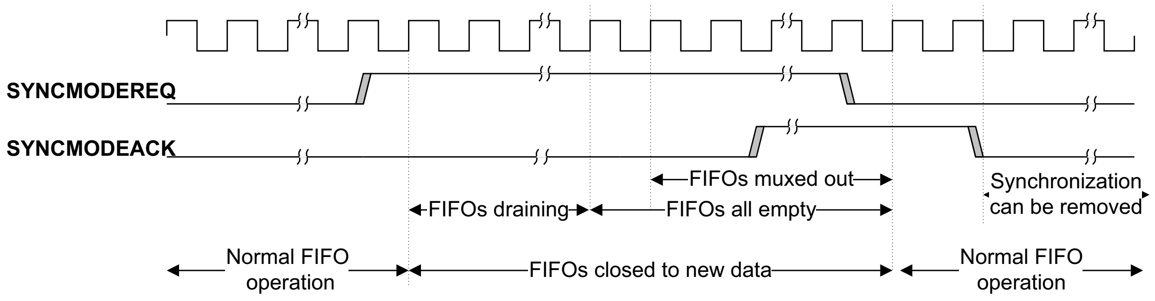


Figure E-3 IEC request/acknowledge interface

Figure E-4 shows the AXI write channel going from VCORE to VSOC. In asynchronous mode data outputs come from FIFO slot 0 and FIFO slot 1 according to the read and write values of the FIFO pointers. Bypass data is clamped to zero. When **SYNCMODEREQ** goes HIGH the IEM slice closes its FIFOs to new data. Source data is driven to output data through the bypass data channel. **SYNCMODEACK** is driven HIGH to acknowledge the synchronous mode request.

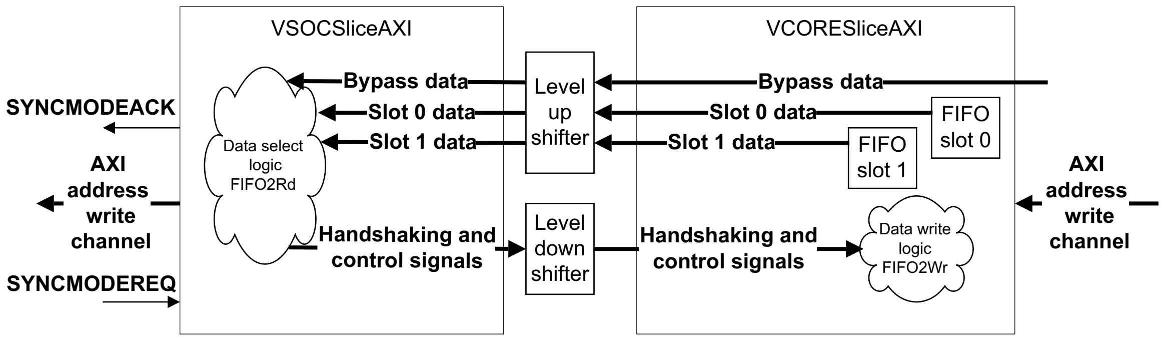


Figure E-4 AXI write channel

Glossary

This glossary describes some of the terms used in ARM manuals. Where terms can have several meanings, the meaning presented here is intended.

Abort	A mechanism that indicates to a core that the value associated with a memory access is invalid. An abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction or data memory. An abort is classified as either a Prefetch or Data Abort, and an internal or External Abort. *See also* Data Abort, External Abort and Prefetch Abort.
Abort model	An abort model is the defined behavior of an ARM processor in response to a Data Abort exception. Different abort models behave differently with regard to load and store instructions that specify base register write-back.
Addressing modes	A mechanism, shared by many different instructions, for generating values used by the instructions. For four of the ARM addressing modes, the values generated are memory addresses (which is the traditional role of an addressing mode). A fifth addressing mode generates values to be used as operands by data-processing instructions.

Advanced eXtensible Interface (AXI)

A bus protocol that supports separate address/control and data phases, unaligned data transfers using byte strobes, burst-based transactions with only start address issued, separate read and write data channels to enable low-cost DMA, ability to issue multiple outstanding addresses, out-of-order transaction completion, and easy addition of register stages to provide timing closure.

The AXI protocol also includes optional extensions to cover signaling for low-power operation.

AXI is targeted at high performance, high clock frequency system designs and includes a number of features that make it very suitable for high speed sub-micron interconnect.

Advanced High-performance Bus (AHB)

A bus protocol with a fixed pipeline between address/control and data phases. It only supports a subset of the functionality provided by the AMBA AXI protocol. The full AMBA AHB protocol specification includes a number of features that are not commonly required for master and slave IP developments and ARM Limited recommends only a subset of the protocol is usually used. This subset is defined as the AMBA AHB-Lite protocol.

See also Advanced Microcontroller Bus Architecture and AHB-Lite.

Advanced Microcontroller Bus Architecture (AMBA)

A family of protocol specifications that describe a strategy for the interconnect. AMBA is the ARM open standard for on-chip buses. It is an on-chip bus specification that details a strategy for the interconnection and management of functional blocks that make up a *System-on-Chip* (SoC). It aids in the development of embedded processors with one or more CPUs or signal processors and multiple peripherals. AMBA complements a reusable design methodology by defining a common backbone for SoC modules.

Advanced Peripheral Bus (APB)

A simpler bus protocol than AXI and AHB. It is designed for use with ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. Connection to the main system bus is through a system-to-peripheral bus bridge that helps to reduce system power consumption.

AHB *See* Advanced High-performance Bus.

AHB Access Port (AHB-AP)

An optional component of the DAP that provides an AHB interface to a SoC.

AHB-AP *See* AHB Access Port.

AHB-Lite	A subset of the full AMBA AHB protocol specification. It provides all of the basic functions required by the majority of AMBA AHB slave and master designs, particularly when used with a multi-layer AMBA interconnect. In most cases, the extra facilities provided by a full AMBA AHB interface are implemented more efficiently by using an AMBA AXI protocol interface.
Aligned	A data item stored at an address that is divisible by the number of bytes that defines the data size is said to be aligned. Aligned words and halfwords have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore stipulate addresses that are divisible by four and two respectively.
AMBA	*See* Advanced Microcontroller Bus Architecture.
Advanced Trace Bus (ATB)	A bus used by trace devices to share CoreSight capture resources.
APB	*See* Advanced Peripheral Bus.
Application Specific Integrated Circuit (ASIC)	An integrated circuit that has been designed to perform a specific application function. It can be custom-built or mass-produced.
Application Specific Standard Part/Product (ASSP)	An integrated circuit that has been designed to perform a specific application function. Usually consists of two or more separate circuit functions combined as a building block suitable for use in a range of products for one or more specific application markets.
Architecture	The organization of hardware and/or software that characterizes a processor and its attached components, and enables devices with similar characteristics to be grouped together when describing their behavior, for example, Harvard architecture, instruction set architecture, ARMv6 architecture.
Arithmetic instruction	Any VFPv2 *Coprocessor Data Processing* (CDP) instruction except FCPY, FABS, and FNEG. *See also* CDP instruction.
ARM instruction	A word that specifies an operation for an ARM processor to perform. ARM instructions must be word-aligned.
ARM state	A processor that is executing ARM (32-bit) word-aligned instructions is operating in ARM state.
ASIC	*See* Application Specific Integrated Circuit.
ASSP	*See* Application Specific Standard Part/Product.
ATB	*See* Advanced Trace Bus.

ATB bridge

A synchronous ATB bridge provides a register slice to facilitate timing closure through the addition of a pipeline stage. It also provides a unidirectional link between two synchronous ATB domains.

An asynchronous ATB bridge provides a unidirectional link between two ATB domains with asynchronous clocks. It is intended to support connection of components with ATB ports residing in different clock domains.

ATPG

See Automatic Test Pattern Generation.

Automatic Test Pattern Generation (ATPG)

The process of automatically generating manufacturing test vectors for an ASIC design, using a specialized software tool.

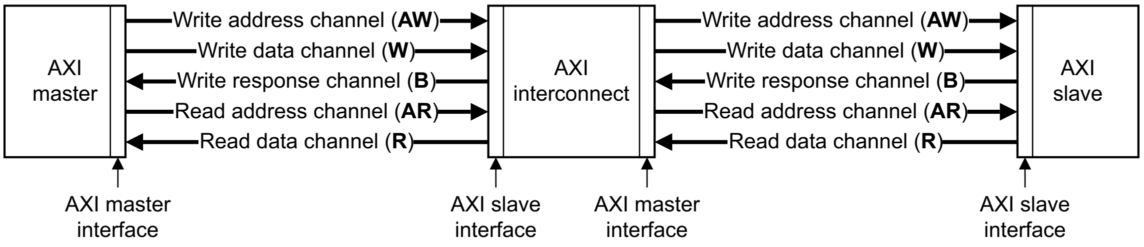
AXI

See Advanced eXtensible Interface.

AXI channel order and interfaces

The block diagram shows:

- the order in which AXI channel signals are described
- the master and slave interface conventions for AXI components.



AXI terminology

The following AXI terms are general. They apply to both masters and slaves:

Active read transaction

A transaction for which the read address has transferred, but the last read data has not yet transferred.

Active transfer

A transfer for which the **xVALID**[1] handshake has asserted, but for which **xREADY** has not yet asserted.

1. The letter **x** in the signal name denotes an AXI channel as follows:

AW	Write address channel.
W	Write data channel.
B	Write response channel.
AR	Read address channel.
R	Read data channel.

Active write transaction

A transaction for which the write address or leading write data has transferred, but the write response has not yet transferred.

Completed transfer

A transfer for which the **xVALID/xREADY** handshake is complete.

Payload The non-handshake signals in a transfer.

Transaction An entire burst of transfers, comprising an address, one or more data transfers and a response transfer (writes only).

Transmit An initiator driving the payload and asserting the relevant **xVALID** signal.

Transfer A single exchange of information. That is, with one **xVALID/xREADY** handshake.

The following AXI terms are master interface attributes. To obtain optimum performance, they must be specified for all components with an AXI master interface:

Combined issuing capability

The maximum number of active transactions that a master interface can generate. This is specified instead of write or read issuing capability for master interfaces that use a combined storage for active write and read transactions.

Read ID capability

The maximum number of different **ARID** values that a master interface can generate for all active read transactions at any one time.

Read ID width

The number of bits in the **ARID** bus.

Read issuing capability

The maximum number of active read transactions that a master interface can generate.

Write ID capability

The maximum number of different **AWID** values that a master interface can generate for all active write transactions at any one time.

Write ID width

The number of bits in the **AWID** and **WID** buses.

Write interleave capability

The number of active write transactions for which the master interface is capable of transmitting data. This is counted from the earliest transaction.

Write issuing capability

The maximum number of active write transactions that a master interface can generate.

The following AXI terms are slave interface attributes. To obtain optimum performance, they must be specified for all components with an AXI slave interface

Combined acceptance capability

The maximum number of active transactions that a slave interface can accept. This is specified instead of write or read acceptance capability for slave interfaces that use a combined storage for active write and read transactions.

Read acceptance capability

The maximum number of active read transactions that a slave interface can accept.

Read data reordering depth

The number of active read transactions for which a slave interface can transmit data. This is counted from the earliest transaction.

Write acceptance capability

The maximum number of active write transactions that a slave interface can accept.

Write interleave depth

The number of active write transactions for which the slave interface can receive data. This is counted from the earliest transaction.

Back-annotation The process of applying timing characteristics from the implementation process onto a model.

Banked registers Those physical registers whose use is defined by the current processor mode. The banked registers are r8 to r14.

Base register A register specified by a load or store instruction that is used to hold the base value for the instruction’s address calculation. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

Base register write-back

Updating the contents of the base register used in an instruction target address calculation so that the modified address is changed to the next higher or lower sequential address in memory. This means that it is not necessary to fetch the target address for successive instruction transfers and enables faster burst accesses to sequential memory.

Beat

Alternative word for an individual transfer within a burst. For example, an INCR4 burst comprises four beats.

See also Burst.

BE-8

Big-endian view of memory in a byte-invariant system.

See also BE-32, LE, Byte-invariant and Word-invariant.

BE-32

Big-endian view of memory in a word-invariant system.

See also BE-8, LE, Byte-invariant and Word-invariant.

Big-endian

Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory.

See also Little-endian and Endianness.

Big-endian memory

Memory in which:

- a byte or halfword at a word-aligned address is the most significant byte or halfword within the word at that address
- a byte at a halfword-aligned address is the most significant byte within the halfword at that address.

See also Little-endian memory.

Block address

An address that comprises a tag, an index, and a word field. The tag bits identify the way that contains the matching cache entry for a cache hit. The index bits identify the set being addressed. The word field contains the word address that can be used to identify specific words, halfwords, or bytes within the cache entry.

See also Cache terminology diagram on the last page of this glossary.

Bounce

The VFP coprocessor bounces an instruction when it fails to signal the acceptance of a valid VFP instruction to the ARM processor. This action initiates Undefined instruction processing by the ARM processor. The VFP support code is called to complete the instruction that was found to be exceptional or unsupported by the VFP coprocessor.

See also Trigger instruction, Potentially exceptional instruction, and Exceptional state.

Boundary scan chain	A boundary scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.
Branch folding	Branch folding is a technique where, on the prediction of most branches, the branch instruction is completely removed from the instruction stream presented to the execution pipeline. Branch folding can significantly improve the performance of branches, taking the CPI for branches below one.
Branch phantom	The condition codes of a predicted taken branch.
Branch prediction	The process of predicting if conditional branches are to be taken or not in pipelined processors. Successfully predicting if branches are to be taken enables the processor to prefetch the instructions following a branch before the condition is fully resolved. Branch prediction can be done in software or by using custom hardware. Branch prediction techniques are categorized as static, in which the prediction decision is decided before run time, and dynamic, in which the prediction decision can change during program execution.
Breakpoint	A breakpoint is a mechanism provided by debuggers to identify an instruction at which program execution is to be halted. Breakpoints are inserted by the programmer to enable inspection of register contents, memory locations, variable values at fixed points in the program execution to test that the program is operating correctly. Breakpoints are removed after the program is successfully tested. *See also* Watchpoint.
Burst	A group of transfers to consecutive addresses. Because the addresses are consecutive, there is no requirement to supply an address for any of the transfers after the first one. This increases the speed at which the group of transfers can occur. Bursts over AHB buses are controlled using the **HBURST** signals to specify if transfers are single, four-beat, eight-beat, or 16-beat bursts, and to specify how the addresses are incremented. *See also* Beat.
Byte	An 8-bit data item.
Byte-invariant	In a byte-invariant system, the address of each byte of memory remains unchanged when switching between little-endian and big-endian operation. When a data item larger than a byte is loaded from or stored to memory, the bytes making up that data item are arranged into the correct order depending on the endianness of the memory access.

	The ARM architecture supports byte-invariant systems in ARMv6 and later versions. When byte-invariant support is selected, unaligned halfword and word memory accesses are also supported. Multi-word accesses are expected to be word-aligned. *See also* Word-invariant.
Byte lane strobe	An AHB signal, **HBSTRB**, that is used for unaligned or mixed-endian data accesses to determine which byte lanes are active in a transfer. One bit of **HBSTRB** corresponds to eight bits of the data bus.
Byte swizzling	The reverse ordering of bytes in a word.
Cache	A block of on-chip or off-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions and/or data. This is done to greatly reduce the average speed of memory accesses and so to increase processor performance. *See also* Cache terminology diagram on the last page of this glossary.
Cache contention	When the number of frequently-used memory cache lines that use a particular cache set exceeds the set-associativity of the cache. In this case, main memory activity increases and performance decreases.
Cache hit	A memory access that can be processed at high speed because the instruction or data that it addresses is already held in the cache.
Cache line	The basic unit of storage in a cache. It is always a power of two words in size (usually four or eight words), and is required to be aligned to a suitable memory boundary. *See also* Cache terminology diagram on the last page of this glossary.
Cache line index	The number associated with each cache line in a cache way. Within each cache way, the cache lines are numbered from 0 to (set associativity) -1. *See also* Cache terminology diagram on the last page of this glossary.
Cache lockdown	To fix a line in cache memory so that it cannot be overwritten. Cache lockdown enables critical instructions and/or data to be loaded into the cache so that the cache lines containing them are not subsequently reallocated. This ensures that all subsequent accesses to the instructions/data concerned are cache hits, and therefore complete as quickly as possible.
Cache miss	A memory access that cannot be processed at high speed because the instruction/data it addresses is not in the cache and a main memory access is required.
Cache set	A cache set is a group of cache lines (or blocks). A set contains all the ways that can be addressed with the same index. The number of cache sets is always a power of two. *See also* Cache terminology diagram on the last page of this glossary.

Cache way	A group of cache lines (or blocks). It is 2 to the power of the number of index bits in size. *See also* Cache terminology diagram on the last page of this glossary.
CAM	*See* Content Addressable Memory.
Cast out	*See* Victim.
CDP instruction	Coprocessor data processing instruction. For the VFP11 coprocessor, CDP instructions are arithmetic instructions and FCPY, FABS, and FNEG. *See also* Arithmetic instruction.
Clean	A cache line that has not been modified while it is in the cache is said to be clean. To clean a cache is to write dirty cache entries into main memory. If a cache line is clean, it is not written on a cache miss because the next level of memory contains the same data as the cache. *See also* Dirty.
Clock gating	Gating a clock signal for a macrocell with a control signal and using the modified clock that results to control the operating state of the macrocell.
Clocks Per Instruction (CPI)	*See* Cycles Per Instruction (CPI).
Coherency	*See* Memory coherency.
Cold reset	Also known as power-on reset. Starting the processor by turning power on. Turning power off and then back on again clears main memory and many internal settings. Some program failures can lock up the processor and require a cold reset to enable the system to be used again. In other cases, only a warm reset is required. *See also* Warm reset.
Communications channel	The hardware used for communicating between the software running on the processor, and an external host, using the debug interface. When this communication is for debug purposes, it is called the Debug Comms Channel. In an ARMv6 compliant core, the communications channel includes the Data Transfer Register, some bits of the Data Status and Control Register, and the external debug interface controller, such as the DBGTAP controller in the case of the JTAG interface.
Condition field	A four-bit field in an instruction that specifies a condition under which the instruction can execute.

Conditional execution
If the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing.

Context
The environment that each process operates in for a multitasking operating system. In ARM processors, this is limited to mean the Physical Address range that it can access in memory and the associated memory access permissions.

See also Fast context switch.

Control bits
The bottom eight bits of a Program Status Register (PSR). The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.

Coprocessor
A processor that supplements the main processor. It carries out additional functions that the main processor cannot perform. Usually used for floating-point math calculations, signal processing, or memory management.

Copy back
See Write-back.

Core
A core is that part of a processor that contains the ALU, the datapath, the general-purpose registers, the Program Counter, and the instruction decode and control circuitry.

Core module
In the context of an ARM Integrator, a core module is an add-on development board that contains an ARM processor and local memory. Core modules can run standalone, or can be stacked onto Integrator motherboards.

Core reset
See Warm reset.

CPI
See Cycles per instruction.

CPSR
See Current Program Status Register

Current Program Status Register (CPSR)
The register that holds the current operating processor status.

Cycles Per instruction (CPI)
Cycles per instruction (or clocks per instruction) is a measure of the number of computer instructions that can be performed in one clock cycle. This figure of merit can be used to compare the performance of different CPUs that implement the same instruction set against each other. The lower the value, the better the performance.

Data Abort
An indication from a memory system to a core that it must halt execution of an attempted illegal memory access. A Data Abort is attempting to access invalid data memory.

See also Abort, External Abort, and Prefetch Abort.

Data cache
A block of on-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used data. This is done to greatly reduce the average speed of memory accesses and so to increase processor performance.

DBGTAP
See Debug Test Access Port.

Debug Access Port (DAP)
A TAP block that acts as an AMBA (AHB or AHB-Lite) master for access to a system bus. The DAP is the term used to encompass a set of modular blocks that support system wide debug. The DAP is a modular component, intended to be extendable to support optional access to multiple systems such as memory mapped AHB and CoreSight APB through a single debug interface.

Debugger
A debugging system that includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.

Debug Test Access Port (DBGTAP)
The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **DBGTDI**, **DBGTDO**, **DBGTMS**, and **TCK**. The optional terminal is **TRST**. This signal is mandatory in ARM cores because it is used to reset the debug logic.

Default NaN mode
A mode in which all operations that result in a NaN return the default NaN, regardless of the cause of the NaN result. This mode is compliant with the IEEE 754 standard but implies that all information contained in any input NaNs to an operation is lost.

Denormalized value
See Subnormal value.

Design Simulation Model (DSM)
A functional simulation model of the device that is derived from the *Register Transfer Level* (RTL) but which does not reveal its internal structure. The DSM does not model any features added during synthesis such as internal scan chains. The DSM provides higher speed for functional simulation than that of the Sign-Off Model (SOM).

Device Validation Suite (DVS)
A set of tests to check the functionality of a device against the functionality defined in the Technical Reference Manual. Also stresses *Bus Interface Unit* (BIU), and low-level memory sub-system, pipeline, and cache behavior.

Direct-mapped cache
A one-way set-associative cache. Each cache set consists of a single cache line, so cache lookup selects and checks a single cache line.

Dirty	A cache line in a write-back cache that has been modified while it is in the cache is said to be dirty. A cache line is marked as dirty by setting the dirty bit. If a cache line is dirty, it must be written to memory on a cache miss because the next level of memory contains data that has not been updated. The process of writing dirty data to main memory is called cache cleaning. *See also* Clean.
Disabled exception	An exception is disabled when its exception enable bit in the FPCSR is not set. For these exceptions, the IEEE 754 standard defines the result to be returned. An operation that generates an exception condition can bounce to the support code to produce the result defined by the IEEE 754 standard. The exception is not reported to the user trap handler.
DNM	*See* Do Not Modify.
Do Not Modify (DNM)	In Do Not Modify fields, the value must not be altered by software. DNM fields read as Unpredictable values, and must only be written with the same value read from the same field on the same processor. DNM fields are sometimes followed by RAZ or RAO in parentheses to show which way the bits should read for future compatibility, but programmers must not rely on this behavior.
Double-precision value	Consists of two 32-bit words that must appear consecutively in memory and must both be word-aligned, and that is interpreted as a basic double-precision floating-point number according to the IEEE 754-1985 standard.
Doubleword	A 64-bit data item. The contents are taken as being an unsigned integer unless otherwise stated.
Doubleword-aligned	A data item having a memory address that is divisible by eight.
DSM	*See* Design Simulation Model.
DVS	*See* Device Validation Suite.
EmbeddedICE logic	An on-chip logic block that provides TAP-based debug support for ARM processor cores. It is accessed through the TAP controller on the ARM core using the JTAG interface.
EmbeddedICE-RT	The JTAG-based hardware provided by debuggable ARM processors to aid debugging in real-time.

Enabled exception	An exception is enabled when its exception enable bit in the FPCSR is set. When an enabled exception occurs, a trap to the user handler is taken. An operation that generates an exception condition might bounce to the support code to produce the result defined by the IEEE 754 standard. The exception is then reported to the user trap handler.
Endianness	Byte ordering. The scheme that determines the order in which successive bytes of a data word are stored in memory. An aspect of the system's memory mapping. *See also* Little-endian and Big-endian
Exception	A fault or error event that is considered serious enough to require that program execution is interrupted. Examples include attempting to perform an invalid memory access, external interrupts, and undefined instructions. When an exception occurs, normal program flow is interrupted and execution is resumed at the corresponding exception vector. This contains the first instruction of the interrupt handler to deal with the exception.
Exceptional state	When a potentially exceptional instruction is issued, the VFP11 coprocessor sets the EX bit, FPEXC[31], and loads a copy of the potentially exceptional instruction in the FPINST register. If the instruction is a short vector operation, the register fields in FPINST are altered to point to the potentially exceptional iteration. When in the exceptional state, the issue of a trigger instruction to the VFP11 coprocessor causes a bounce. *See also* Bounce, Potentially exceptional instruction, and Trigger instruction.
Exception service routine	*See* Interrupt handler.
Exception vector	*See* Interrupt vector.
Exponent	The component of a floating-point number that normally signifies the integer power to which two is raised in determining the value of the represented number.
External Abort	An indication from an external memory system to a core that it must halt execution of an attempted illegal memory access. An External Abort is caused by the external memory system as a result of attempting to access invalid memory. *See also* Abort, Data Abort and Prefetch Abort.
eXternal Verification Component (XVC)	A model that is used to provide system/device stimulus and monitor responses. *See also* XVC Test Scenario Manager.

Fast context switch

In a multitasking system, the point at which the time-slice allocated to one process stops and the one for the next process starts. If processes are switched often enough, they can appear to a user to be running in parallel, as well as being able to respond quicker to external events that might affect them.

In ARM processors, a fast context switch is caused by the selection of a non-zero PID value to switch the context to that of the next process. A fast context switch causes each Virtual Address for a memory access, generated by the ARM processor, to produce a Modified Virtual Address which is sent to the rest of the memory system to be used in place of a normal Virtual Address. For some cache control operations Virtual Addresses are passed to the memory system as data. In these cases no address modification takes place.

See also Fast Context Switch Extension.

Fast Context Switch Extension (FCSE)

An extension to the ARM architecture that enables cached processors with an MMU to present different addresses to the rest of the memory system for different software processes, even when those processes are using identical addresses.

See also Fast context switch.

FCSE

See Fast Context Switch Extension.

Fd

The destination register and the accumulate value in triadic operations. Sd for single-precision operations and Dd for double-precision.

Flash Patch and Breakpoint unit (FPB)

A set of address matching tags, which reroute accesses into flash to a special part of SRAM. This allows patching flash locations for breakpointing and quick fixes or changes.

Flat address mapping

A system of organizing memory in which each Physical Address contained within the memory space is the same as its corresponding Virtual Address.

Flush-to-zero mode

In this mode, the VFP11 coprocessor treats the following values as positive zeros:

- arithmetic operation inputs that are in the subnormal range for the input precision
- arithmetic operation results, other than computed zero results, that are in the subnormal range for the input precision before rounding.

The VFP11 coprocessor does not interpret these values as subnormal values or convert them to subnormal values.

The subnormal range for the input precision is $-2^{Emin} < x < 0$ or $0 < x < 2^{Emin}$.

Fm	The second source operand in dyadic or triadic operations. Sm for single-precision operations and Dm for double-precision
Fn	The first source operand in dyadic or triadic operations. Sn for single-precision operations and Dn for double-precision.
Formatter	The formatter is an internal input block in the ETB and TPIU that embeds the trace source ID within the data to create a single trace stream.
Fraction	The floating-point field that lies to the right of the implied binary point.
Front of queue pointer	Pointer to the next entry to be written to in the write buffer.
Fully-associative cache	A cache that has just one cache set that consists of the entire cache. The number of cache entries is the same as the number of cache ways. *See also* Direct-mapped cache.
Gray code	Continuous binary code in which only one bit changes for a change to the next state up or down.
Half-rate clocking	Dividing the trace clock by two so that the TPA can sample trace data signals on both the rising and falling edges of the trace clock. The primary purpose of half-rate clocking is to reduce the signal transition rate on the trace clock of an ASIC for very high-speed systems.
Halfword	A 16-bit data item.
Halt mode	One of two mutually exclusive debug modes. In halt mode all processor execution halts when a breakpoint or watchpoint is encountered. All processor state, coprocessor state, memory and input/output locations can be examined and altered by the JTAG interface. *See also* Monitor debug-mode.
High vectors	Alternative locations for exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.
Hit-Under-Miss (HUM)	A buffer that enables program execution to continue, even though there has been a data miss in the cache.
Host	A computer that provides data and other services to another computer. Especially, a computer providing debugging services to a target being debugged.
HUM	*See* Hit-Under-Miss.

IEEE 754 standard	*IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985*. The standard that defines data types, correct operation, exception types and handling, and error bounds for floating-point systems. Most processors are built in compliance with the standard in either hardware or a combination of hardware and software.
IEM	*See* Intelligent Energy Manager.
IGN	*See* Ignore.
Ignore (IGN)	Must ignore memory writes.
Illegal instruction	An instruction that is architecturally Undefined.
IMB	*See* Instruction Memory Barrier.
Implementation-defined	Means that the behavior is not architecturally defined, but should be defined and documented by individual implementations.
Implementation-specific	Means that the behavior is not architecturally defined, and does not have to be documented by individual implementations. Used when there are a number of implementation options available and the option chosen does not affect software compatibility.
Imprecise tracing	A filtering configuration where instruction or data tracing can start or finish earlier or later than expected. Most cases cause tracing to start or finish later than expected. For example, if **TraceEnable** is configured to use a counter so that tracing begins after the fourth write to a location in memory, the instruction that caused the fourth write is not traced, although subsequent instructions are. This is because the use of a counter in the **TraceEnable** configuration always results in imprecise tracing.
Index	*See* Cache index.
Index register	A register specified in some load or store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address, which is sent to memory. Some addressing modes optionally enable the index register value to be shifted prior to the addition or subtraction.
Infinity	In the IEEE 754 standard format to represent infinity, the exponent is the maximum for the precision and the fraction is all zeros.
Input exception	A VFP exception condition in which one or more of the operands for a given operation are not supported by the hardware. The operation bounces to support code for processing.

Instruction cache	A block of on-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions. This is done to greatly reduce the average speed of memory accesses and so to increase processor performance.
Instruction cycle count	The number of cycles for which an instruction occupies the Execute stage of the pipeline.
Instruction Memory Barrier (IMB)	An operation to ensure that the prefetch buffer is flushed of all out-of-date instructions.
Intelligent Energy Manager (IEM)	A technology that enables dynamic voltage scaling and clock frequency variation to be used to reduce power consumption in a device.
Intermediate result	An internal format used to store the result of a calculation before rounding. This format can have a larger exponent field and fraction field than the destination format.
Internal scan chain	A series of registers connected together to form a path through a device, used during production testing to import test patterns into internal nodes of the device and export the resulting values.
Interrupt handler	A program that control of the processor is passed to when an interrupt occurs.
Interrupt vector	One of a number of fixed addresses in low memory, or in high memory if high vectors are configured, that contains the first instruction of the corresponding interrupt handler.
Invalidate	To mark a cache line as being not valid by clearing the valid bit. This must be done whenever the line does not contain a valid cache entry. For example, after a cache flush all lines are invalid.
Jazelle architecture	The ARM Jazelle architecture extends the Thumb and ARM operating states by adding a Java state to the processor. Instruction set support for entering and exiting Java applications, real-time interrupt handling, and debug support for mixed Java/ARM applications is present. When in Java state, the processor fetches and decodes Java bytecodes and maintains the Java operand stack.
Joint Test Action Group (JTAG)	The name of the organization that developed standard IEEE 1149.1. This standard defines a boundary-scan architecture used for in-circuit testing of integrated circuit devices. It is commonly known by the initials JTAG.
JTAG	*See* Joint Test Action Group.
LE	Little endian view of memory in both byte-invariant and word-invariant systems. See also Byte-invariant, Word-invariant.

Line	*See* Cache line.
Little-endian	Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory. *See also* Big-endian and Endianness.
Little-endian memory	Memory in which: - a byte or halfword at a word-aligned address is the least significant byte or halfword within the word at that address - a byte at a halfword-aligned address is the least significant byte within the halfword at that address. *See also* Big-endian memory.
Load/store architecture	A processor architecture where data-processing operations only operate on register contents, not directly on memory contents.
Load Store Unit (LSU)	The part of a processor that handles load and store transfers.
LSU	*See* Load Store Unit.
Macrocell	A complex logic block with a defined interface and behavior. A typical VLSI system comprises several macrocells (such as a processor, an ETM, and a memory block) plus application-specific logic.
Memory bank	One of two or more parallel divisions of interleaved memory, usually one word wide, that enable reads and writes of multiple words at a time, rather than single words. All memory banks are addressed simultaneously and a bank enable or chip select signal determines which of the banks is accessed for each transfer. Accesses to sequential word addresses cause accesses to sequential banks. This enables the delays associated with accessing a bank to occur during the access to its adjacent bank, speeding up memory transfers.
Memory coherency	A memory is coherent if the value read by a data read or instruction fetch is the value that was most recently written to that location. Memory coherency is made difficult when there are multiple possible physical locations that are involved, such as a system that has main memory, a write buffer and a cache.
Memory Management Unit (MMU)	Hardware that controls caches and access permissions to blocks of memory, and translates virtual addresses to physical addresses.

Memory Protection Unit (MPU)	Hardware that controls access permissions to blocks of memory. Unlike an MMU, an MPU does not translate virtual addresses to physical addresses.
Microprocessor	*See* Processor.
Miss	*See* Cache miss.
MMU	*See* Memory Management Unit.
Model Manager	A software control manager that handles the event transactions between the model and simulator.
Modified Virtual Address (MVA)	A Virtual Address produced by the ARM processor can be changed by the current Process ID to provide a *Modified Virtual Address* (MVA) for the MMUs and caches. *See also* Fast Context Switch Extension.
Monitor debug-mode	One of two mutually exclusive debug modes. In Monitor debug-mode the processor enables a software abort handler provided by the debug monitor or operating system debug task. When a breakpoint or watchpoint is encountered, this enables vital system interrupts to continue to be serviced while normal program execution is suspended. *See also* Halt mode.
MPU	*See* Memory Protection Unit.
Multi-ICE	A JTAG-based tool for debugging embedded systems.
Multi-layered	An AMBA scheme to break a bus into segments that are controlled in access. This enables local masters to reduce lock overhead.
Multi master	An AMBA bus sharing scheme (not in AMBA Lite) where different masters can gain a bus lock (Grant) to access the bus in an interleaved fashion.
MVA	*See* Modified Virtual Address.
NaN	Not a number. A symbolic entity encoded in a floating-point format that has the maximum exponent field and a nonzero fraction. An SNaN causes an invalid operand exception if used as an operand and a most significant fraction bit of zero. A QNaN propagates through almost every arithmetic operation without signaling exceptions and has a most significant fraction bit of one.
PA	*See* Physical Address.
Penalty	The number of cycles in which no useful Execute stage pipeline activity can occur because an instruction flow is different from that assumed or predicted.

Potentially exceptional instruction

An instruction that is determined, based on the exponents of the operands and the sign bits, to have the potential to produce an overflow, underflow, or invalid condition. After this determination is made, the instruction that has the potential to cause an exception causes the VFP11 coprocessor to enter the exceptional state and bounce the next trigger instruction issued.

See also Bounce, Trigger instruction, and Exceptional state.

Power-on reset *See* Cold reset.

Prefetching In pipelined processors, the process of fetching instructions from memory to fill up the pipeline before the preceding instructions have finished executing. Prefetching an instruction does not mean that the instruction must be executed.

Prefetch Abort An indication from a memory system to a core that it must halt execution of an attempted illegal memory access. A Prefetch Abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction memory.

See also Data Abort, External Abort and Abort.

Processor A processor is the circuitry in a computer system required to process data using the computer instructions. It is an abbreviation of microprocessor. A clock source, power supplies, and main memory are also required to create a minimum complete working computer system.

Programming Language Interface (PLI)

For Verilog simulators, an interface by which so-called foreign code (code written in a different language) can be included in a simulation.

Physical Address (PA)

The MMU performs a translation on *Modified Virtual Addresses* (MVA) to produce the *Physical Address* (PA) which is given to AHB to perform an external access. The PA is also stored in the data cache to avoid the necessity for address translation when data is cast out of the cache.

See also Fast Context Switch Extension.

Read Reads are defined as memory operations that have the semantics of a load. That is, the ARM instructions LDM, LDRD, LDC, LDR, LDRT, LDRSH, LDRH, LDRSB, LDRB, LDRBT, LDREX, RFE, STREX, SWP, and SWPB, and the Thumb instructions LDM, LDR, LDRSH, LDRH, LDRSB, LDRB, and POP. Java instructions that are accelerated by hardware can cause a number of reads to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

RealView ICE A system for debugging embedded processor cores using a JTAG interface.

Region A partition of instruction or data memory space.

Remapping Changing the address of physical memory or devices after the application has started executing. This is typically done to enable RAM to replace ROM when the initialization has been completed.

Replicator A replicator enables two trace sinks to be wired together and to operate independently on the same incoming trace stream. The input trace stream is output onto two (independent) ATB ports.

Reserved A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces Unpredictable results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as 0 and read as 0.

Rounding mode The IEEE 754 standard requires all calculations to be performed as if to an infinite precision. For example, a multiply of two single-precision values must accurately calculate the significand to twice the number of bits of the significand. To represent this value in the destination precision, rounding of the significand is often required. The IEEE 754 standard specifies four rounding modes.

In round-to-nearest mode, the result is rounded at the halfway point, with the tie case rounding up if it would clear the least significant bit of the significand, making it even.

Round-towards-zero mode chops any bits to the right of the significand, always rounding down, and is used by the C, C++, and Java languages in integer conversions.

Round-towards-plus-infinity mode and round-towards-minus-infinity mode are used in interval arithmetic.

RunFast mode In RunFast mode, hardware handles exceptional conditions and special operands. RunFast mode is enabled by enabling default NaN and flush-to-zero modes and disabling all exceptions. In RunFast mode, the VFP11 coprocessor does not bounce to the support code for any legal operation or any operand, but supplies a result to the destination. For all inexact and overflow results and all invalid operations that result from operations not involving NaNs, the result is as specified by the IEEE 754 standard. For operations involving NaNs, the result is the default NaN.

Saved Program Status Register (SPSR)
The register that holds the CPSR of the task immediately before the exception occurred that caused the switch to the current mode.

SBO *See* Should Be One.

SBZ *See* Should Be Zero.

SBZP *See* Should Be Zero or Preserved.

Scalar operation
A VFP coprocessor operation involving a single source register and a single destination register.

See also Vector operation.

Scan chain
A scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.

SCREG
The currently selected scan chain number in an ARM TAP controller.

SDF
See Standard Delay Format.

Set
See Cache set.

Set-associative cache
In a set-associative cache, lines can only be placed in the cache in locations that correspond to the modulo division of the memory address by the number of sets. If there are *n* ways in a cache, the cache is termed *n*-way set-associative. The set-associativity can be any number greater than or equal to 1 and is not restricted to being a power of two.

Short vector operation
A VFP coprocessor operation involving more than one destination register and perhaps more than one source register in the generation of the result for each destination.

Should Be One (SBO)
Should be written as 1 (or all 1s for bit fields) by software. Writing a 0 produces Unpredictable results.

Should Be Zero (SBZ)
Should be written as 0 (or all 0s for bit fields) by software. Writing a 1 produces Unpredictable results.

Should Be Zero or Preserved (SBZP)
Should be written as 0 (or all 0s for bit fields) by software, or preserved by writing the same value back that has been previously read from the same field on the same processor.

Significand
The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of the implied binary point and a fraction field to the right.

Sign-Off Model (SOM)
This is an opaque, compiled simulation model generated from a technology specific netlist of an ARM core, derived after gate level synthesis and timing annotation, that you can use in back-annotated gate-level simulations to prove the function and timing

behavior of the device. It enables accurate timing simulation of SoCs and simulation using production test vectors from Automatic Test Pattern Generation (ATPG) tool such as Synopsys TetraMAX. It only supports back-annotation using SDF files. The SOM includes timing information but provides slower simulation than a DSM.

SOM
See Sign-Off Model.

SPICE
Simulation Program with Integrated Circuit Emphasis. An accurate transistor-level electronic circuit simulation tool that can be used to predict how an equivalent real circuit will behave for given circuit conditions.

SPSR
See Saved Program Status Register

Standard Delay Format (SDF)
The format of a file that contains timing information to the level of individual bits of buses and is used in SDF back-annotation. An SDF file can be generated in a number of ways, but most commonly from a delay calculator.

Stride
The stride field, FPSCR[21:20], specifies the increment applied to register addresses in short vector operations. A stride of 00, specifying an increment of +1, causes a short vector operation to increment each vector register by +1 for each iteration, while a stride of 11 specifies an increment of +2.

Subnormal value
A value in the range ($-2^{Emin} < x < 2^{Emin}$), except for ±0. In the IEEE 754 standard format for single-precision and double-precision operands, a subnormal value has a zero exponent and a nonzero fraction field. The IEEE 754 standard requires that the generation and manipulation of subnormal operands be performed with the same precision as normal operands.

Support code
Software that must be used to complement the hardware to provide compatibility with the IEEE 754 standard. The support code has a library of routines that performs supported functions, such as divide with unsupported inputs or inputs that might generate an exception as well as operations beyond the scope of the hardware. The support code has a set of exception handlers to process exceptional conditions in compliance with the IEEE 754 standard.

Synchronization primitive
The memory synchronization primitive instructions are those instructions that are used to ensure memory synchronization. That is, the LDREX, STREX, SWP, and SWPB instructions.

Tag The upper portion of a block address used to identify a cache line within a cache. The block address from the CPU is compared with each tag in a set in parallel to determine if the corresponding line is in the cache. If it is, it is said to be a cache hit and the line can be fetched from cache. If the block address does not correspond to any of the tags, it is said to be a cache miss and the line must be fetched from the next level of memory.

See also Cache terminology diagram on the last page of this glossary.

TAP *See* Test access port.

Test Access Port (TAP)

The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **TRST**. This signal is mandatory in ARM cores because it is used to reset the debug logic.

Thumb instruction A halfword that specifies an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned.

Thumb state A processor that is executing Thumb (16-bit) halfword aligned instructions is operating in Thumb state.

Tiny A nonzero result or value that is between the positive and negative minimum normal values for the destination precision.

TLB *See* Translation Look-aside Buffer.

Translation Lookaside Buffer (TLB)

A cache of recently used page table entries that avoid the overhead of page table walking on every memory access. Part of the Memory Management Unit.

Translation table A table, held in memory, that contains data that defines the properties of memory areas of various fixed sizes.

Translation table walk

The process of doing a full translation table lookup. It is performed automatically by hardware.

Trap An exceptional condition in a VFP coprocessor that has the respective exception enable bit set in the FPSCR register. The user trap handler is executed.

Trigger instruction The VFP coprocessor instruction that causes a bounce at the time it is issued. A potentially exceptional instruction causes the VFP11 coprocessor to enter the exceptional state. A subsequent instruction, unless it is an FMXR or FMRX instruction accessing the FPEXC, FPINST, or FPSID register, causes a bounce, beginning

	exception processing. The trigger instruction is not necessarily exceptional, and no processing of it is performed. It is retried at the return from exception processing of the potentially exceptional instruction. *See also* Bounce, Potentially exceptional instruction, and Exceptional state.
Undefined	Indicates an instruction that generates an Undefined instruction trap. See the *ARM Architecture Reference Manual* for more details on ARM exceptions.
UNP	*See* Unpredictable.
Unpredictable	Means that the behavior of the ETM cannot be relied upon. Such conditions have not been validated. When applied to the programming of an event resource, only the output of that event resource is Unpredictable. Unpredictable behavior can affect the behavior of the entire system, because the ETM is capable of causing the core to enter debug state, and external outputs may be used for other purposes.
Unpredictable	For reads, the data returned when reading from this location is unpredictable. It can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration. Unpredictable instructions must not halt or hang the processor, or any part of the system.
Unsupported values	Specific data values that are not processed by the VFP coprocessor hardware but bounced to the support code for completion. These data can include infinities, NaNs, subnormal values, and zeros. An implementation is free to select which of these values is supported in hardware fully or partially, or requires assistance from support code to complete the operation. Any exception resulting from processing unsupported data is trapped to user code if the corresponding exception enable bit for the exception is set.
VA	*See* Virtual Address.
Vector operation	A VFP coprocessor operation involving more than one destination register, perhaps involving different source registers in the generation of the result for each destination. *See also* Scalar operation.
Victim	A cache line, selected to be discarded to make room for a replacement cache line that is required as a result of a cache miss. The way in which the victim is selected for eviction is processor-specific. A victim is also known as a cast out.

Virtual Address (VA)

The MMU uses its page tables to translate a Virtual Address into a Physical Address. The processor executes code at the Virtual Address, which might be located elsewhere in physical memory.

See also Fast Context Switch Extension, Modified Virtual Address, and Physical Address.

Warm reset Also known as a core reset. Initializes the majority of the processor excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

Watchpoint A watchpoint is a mechanism provided by debuggers to halt program execution when the data contained by a particular memory address is changed. Watchpoints are inserted by the programmer to allow inspection of register contents, memory locations, and variable values when memory is written to test that the program is operating correctly. Watchpoints are removed after the program is successfully tested. *See also* Breakpoint.

Way *See* Cache way.

WB *See* Write-back.

Word A 32-bit data item.

Word-invariant In a word-invariant system, the address of each byte of memory changes when switching between little-endian and big-endian operation, in such a way that the byte with address A in one endianness has address A EOR 3 in the other endianness. As a result, each aligned word of memory always consists of the same four bytes of memory in the same order, regardless of endianness. The change of endianness occurs because of the change to the byte addresses, not because the bytes are rearranged.

The ARM architecture supports word-invariant systems in ARMv3 and later versions. When word-invariant support is selected, the behavior of load or store instructions that are given unaligned addresses is instruction-specific, and is in general not the expected behavior for an unaligned access. It is recommended that word-invariant systems should use the endianness that produces the desired byte addresses at all times, apart possibly from very early in their reset handlers before they have set up the endianness, and that this early part of the reset handler should use only aligned word memory accesses.

See also Byte-invariant.

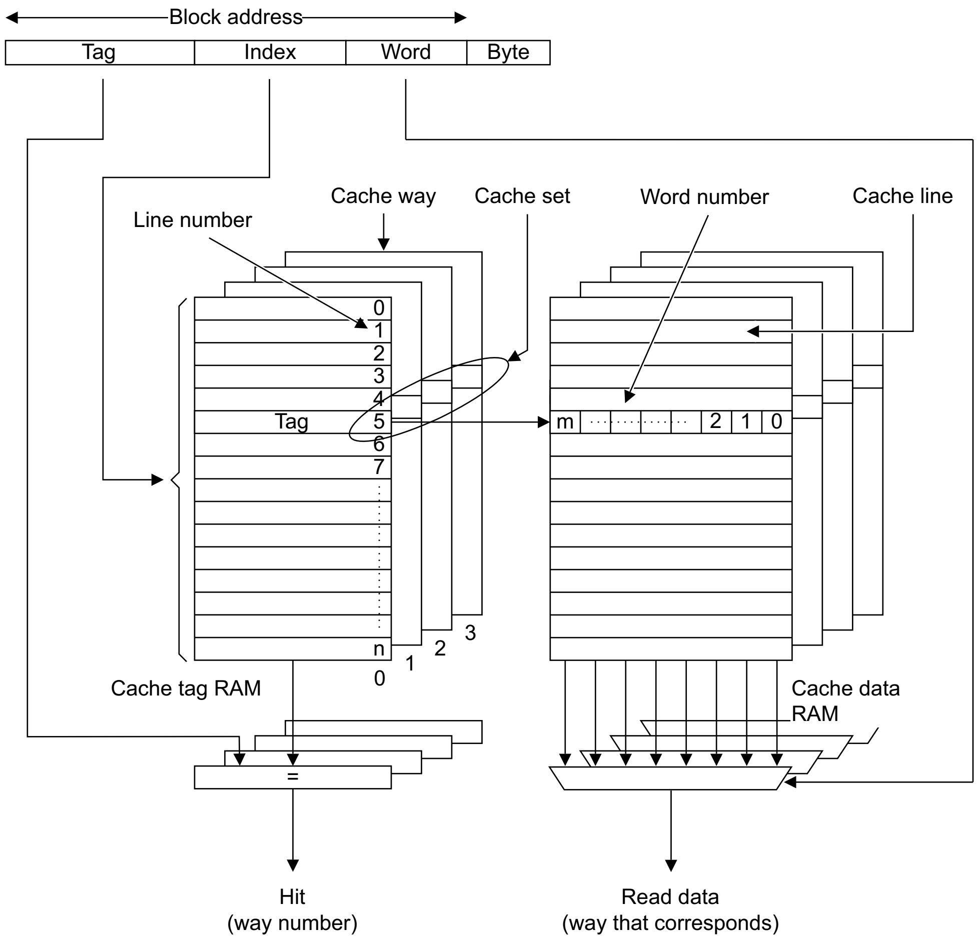
Write Writes are defined as operations that have the semantics of a store. That is, the ARM instructions SRS, STM, STRD, STC, STRT, STRH, STRB, STRBT, STREX, SWP, and SWPB, and the Thumb instructions STM, STR, STRH, STRB, and PUSH. Java instructions that are accelerated by hardware can cause a number of writes to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

Write-back (WB)	In a write-back cache, data is only written to main memory when it is forced out of the cache on line replacement following a cache miss. Otherwise, writes by the processor only update the cache. (Also known as copyback).
Write buffer	A block of high-speed memory, arranged as a FIFO buffer, between the data cache and main memory, whose purpose is to optimize stores to main memory.
Write completion	The memory system indicates to the processor that a write has been completed at a point in the transaction where the memory system is able to guarantee that the effect of the write is visible to all processors in the system. This is not the case if the write is associated with a memory synchronization primitive, or is to a Device or Strongly Ordered region. In these cases the memory system might only indicate completion of the write when the access has affected the state of the target, unless it is impossible to distinguish between having the effect of the write visible and having the state of target updated. This stricter requirement for some types of memory ensures that any side-effects of the memory access can be guaranteed by the processor to have taken place. You can use this to prevent the starting of a subsequent operation in the program order until the side-effects are visible.
Write-through (WT)	In a write-through cache, data is written to main memory at the same time as the cache is updated.
WT	*See* Write-through.
XVC	*See* eXternal Verification Component.
XVC Test Scenario Manager	This co-ordinates the operation of multiple XVCs. *See also* eXternal Verification Component
XTSM	*See* XVC Test Scenario Manager.

Cache terminology diagram

The diagram below illustrates the following cache terminology:

- block address
- cache line
- cache set
- cache way
- index
- tag.


Block address
Tag
Index
Word
Byte
Cache way
Cache set
Word number
Cache line
Line number
0
1
2
3
4
Tag
5
6
7
n
0
1
2
3
m
2
1
0
Cache tag RAM
Cache data RAM
=
Hit (way number)
Read data (way that corresponds)