B3.5 Protected Memory System Architecture, PMSAv7

Supporting a model of unprivileged and privileged software execution requires a memory protection scheme that controls the access rights. Armv7-M supports the *Protected Memory System Architecture* (PMSAv7). The system address space of a PMSAv7 implementation is protected by a *Memory Protection Unit* (MPU). The MPU divides the memory into regions. The number of supported regions is IMPLEMENTATION DEFINED. PMSAv7 can support regions as small as 32 bytes, but the limited register resources in the 4GB address space mean the MPU provides an inherently coarse-grained protection scheme. The scheme is completely predictive, with all control information held in registers that are closely-coupled to the processor. Memory accesses are only required for software control of the MPU register interface, see *Register support for PMSAv7 in the SCS* on page B3-635.

MPU support in Armv7-M is optional.

B3.5.1 Relation of the MPU to the system memory map

When implemented, an MPU's relation to the system memory map described in *The system address map* on page B3-592 is as follows:

- MPU support provides control of access rights on physical addresses. It does not perform address translation.
- When the MPU is disabled or not present, the system adopts the default system memory map listed in Table B3-1 on page B3-592. When the MPU is enabled, the enabled regions define the system address map with the following provisos:
  - Accesses to the *Private Peripheral Bus* (PPB) always use the default system address map.
  - Exception vector reads from the Vector Address Table always use the default system address map.
  - The MPU is restricted in how it can change the default memory map attributes associated with System space, that is, for addresses `0xE0000000` and higher.

    System space is always marked as XN, Execute Never.
  - When the execution priority is less than 0, MPU_CTRL.HFNMIENA determines whether memory accesses use the MPU or the default memory map attributes. The execution priority is less than 0 if the processor is executing the NMI or HardFault handler, or if FAULTMASK is set to 1.
  - The default system memory map can be configured to provide a background region for privileged accesses.
  - Accesses with an address match in more than one region use the highest matching region number for the access attributes.
  - Accesses that do not match all access conditions of a region address match (with the MPU enabled) or a background/default memory map match generate a fault.

B3.5.2 Behavior when the MPU is disabled

Disabling the MPU, by setting the MPU_CTRL.ENABLE bit to 0, means that privileged and unprivileged accesses use the default memory map.

When the MPU is disabled:

- Instruction accesses use the default memory map and attributes shown in Table B3-1 on page B3-592. An access to a memory region with the execute-never attribute generates a MemManage fault, see *Execute Never encoding* on page B3-642. No other permission checks are performed. Additional control of the Cacheability is made by:
  - The CCR.IC bit if separate instruction and data caches are implemented.
  - The CCR.DC bit if unified caches are implemented.
- Data accesses use the default memory map and attributes shown in Table B3-1 on page B3-592. No memory access permission checks are performed, and no aborts can be generated.
- Program flow prediction functions as normal, controlled by the value of the CCR.BP bit.
- Speculative instruction and data fetch operations work as normal, based on the default memory map:
  - Speculative data read operations have no effect if the data cache is disabled.
  - Speculative instruction fetch operations have no effect if the instruction cache is disabled.

B3.5.3 PMSAv7-compliant MPU operation

Armv7-M only supports a unified memory model with respect to MPU region support. All enabled regions provide support for instruction and data accesses.

The base address, size and attributes of a region are all configurable, with the general rule that all regions are naturally aligned. This can be stated as:

`RegionBaseAddress[(N-1):0] = 0, where N is log2(SizeofRegion_in_bytes)`

Memory regions can vary in size as a power of 2. The supported sizes are 2^N, where $5 \leq N \leq 32$. Where there is an overlap between two regions, the register with the highest region number takes priority.

Sub-region support

For regions of 256 bytes or larger, the region can be divided up into eight sub-regions of size $2^{(N-3)}$. Sub-regions within a region can be disabled on an individual basis (8 disable bits) with respect to the associated region attribute register. When a sub-region is disabled, an access match is required from another region, or background matching if enabled. If an access match does not occur a fault is generated. Region sizes below 256 bytes do not support sub-regions, setting MPU_RASR.SRD to non-zero for a region less than 256 bytes is UNPREDICTABLE.

Armv7-M specific support

Armv7-M supports the standard PMSAv7 of the Armv7-R architecture profile, with the following extensions:

- An optimized two register update model, where software can select the region to update by writing to the MPU Region Base Address Register. This optimization applies to the first sixteen memory regions ($0 \leq$ RegionNumber $\leq$ `0xF`) only.
- The MPU Region Base Address Register and the MPU Region Attribute and Size Register pairs are aliased in three consecutive dual-word locations. Using the two register update model, software can modify up to four regions by writing the appropriate even number of words using a single `STM` multi-word store instruction.

MPU pseudocode

The following pseudocode defines the operation of an Armv7-M MPU. The terms used align with the MPU register names and bit field names described in *Register support for PMSAv7 in the SCS* on page B3-635.

```
// ValidateAddress()
// =================

AddressDescriptor ValidateAddress(bits(32) address, AccType acctype, boolean iswrite)
    ispriv = acctype != AccType_UNPRIV && FindPriv();

    AddressDescriptor result;
    Permissions perms;

    result.physicaladdress = address;
    result.memattrs = DefaultMemoryAttributes(address);
    perms = DefaultPermissions(address);

    hit = FALSE;  // assume no valid MPU region and not using default memory map

    isPPBaccess = (address<31:20> == '111000000000');

    if acctype == AccType_VECTABLE || isPPBaccess then
        hit = TRUE;      // use default map for PPB and vector table lookups

    elsif MPU_CTRL.ENABLE == '0' then
        if MPU_CTRL.HFNMIENA == '1' then UNPREDICTABLE;
        else hit = TRUE; // always use default map if MPU disabled

    elsif MPU_CTRL.HFNMIENA == '0' && ExecutionPriority() < 0 then
        hit = TRUE;      // optionally use default for HardFault, NMI and FAULTMASK

    else  // MPU is enabled so check each individual region
```

```
        if (MPU_CTRL.PRIVDEFENA == '1') && ispriv then
            hit = TRUE;  // optional default as background for Privileged accesses

        for r = 0 to (UInt(MPU_TYPE.DREGION) - 1)  // highest matching region wins
            bits(16) size_enable    = MPU_RASR[r]<15:0>;
            bits(32) base_address   = MPU_RBAR[r];
            bits(16) access_control = MPU_RASR[r]<31:16>;

            if size_enable<0> == '1' then  // MPU region enabled so perform checks
                lsbit = UInt(size_enable<5:1>) + 1;
                if lsbit < 5 then UNPREDICTABLE;
                if (lsbit < 8) && (!IsZero(size_enable<15:8>)) then UNPREDICTABLE;

                if lsbit == 32 || address<31:lsbit> == base_address<31:lsbit> then
                    subregion = UInt(address<lsbit-1:lsbit-3>);
                    if size_enable<subregion+8> == '0' then
                        texcb = access_control<5:3,1:0>;
                        S = access_control<2>;
                        perms.ap = access_control<10:8>;
                        perms.xn = access_control<12>;
                        result.memattrs = DefaultTEXDecode(texcb,S);
                        hit = TRUE;

    if address<31:29> == '111' then  // enforce System space execute never
        perms.xn = '1';


    if hit then  // perform check of acquired access permissions
        CheckPermission(perms, address, acctype, iswrite);
    else         // generate fault if no MPU match or use of default not enabled
        if acctype == AccType_IFETCH then
            MMFSR.IACCVIOL = '1';
            MMFSR.MMARVALID = '0';
        else
            MMFSR.DACCVIOL = '1';
            MMAR = address;
            MMFSR.MMARVALID = '1';
        ExceptionTaken(MemManage);

    return result;

// DefaultPermissions()
// ====================

Permissions DefaultPermissions(bits(32) address)

    Permissions perms;

    perms.ap = '011';

    case address<31:29> of
        when '000'
            perms.xn = '0';
        when '001'
            perms.xn = '0';
        when '010'
            perms.xn = '1';
        when '011'
            perms.xn = '0';
        when '100'
            perms.xn = '0';
        when '101'
            perms.xn = '1';
        when '110'
            perms.xn = '1';
        when '111'
            perms.xn = '1';
```

```
    return perms;
```

Access permission checking on page B2-587 defines the `CheckPermission()` function.

MPU fault support

Instruction or data access violations cause a MemManage exception to be generated. See *Fault behavior* on page B1-551 for more details of MemManage exceptions.

B3.5.4 Register support for PMSAv7 in the SCS

Table B3-11 summarizes the register support for a *Memory Protection Unit* (MPU) in the System Control Space. In general and unless otherwise stated, registers support word accesses only, with byte and halfword access UNPREDICTABLE. All MPU register addresses are mapped as little endian.

MPU registers require privileged memory accesses for reads and writes. Any unprivileged access generates a BusFault fault.

There are three general MPU registers:

- The MPU Type Register specified in *MPU Type Register, MPU_TYPE* on page B3-636. This register can be used to determine if an MPU exists, and the number of regions supported.
- The MPU Control Register specified in *MPU Control Register, MPU_CTRL* on page B3-637. The MPU Control Register includes a global enable bit that must be set to 1 to enable the MPU.
- The MPU Region Number Register specified in *MPU Region Number Register, MPU_RNR* on page B3-638.

The MPU Region Number Register selects the associated region registers:

- The MPU Region Base Address Register specified in *MPU Region Base Address Register, MPU_RBAR* on page B3-639.
- The MPU Region Attribute and Size Register to control the region size, sub-region access, access permissions, memory type, and other properties of the memory region in *MPU Region Attribute and Size Register, MPU_RASR* on page B3-640.

Each set of region registers includes its own region enable bit.

If an Armv7-M implementation does not support PMSAv7, only the MPU Type Register is required. The MPU Control Register is RAZ/WI, and all other registers in this region are reserved, UNK/SBZP.

All MPU registers are 32-bits wide.

Table B3-11 MPU register summary

Address	Name	Type	Reset	Description
0xE000ED90	MPU_TYPE	RO	IMPLEMENTATION DEFINED	*MPU Type Register, MPU_TYPE* on page B3-636
0xE000ED94	MPU_CTRL	RW	0x00000000	*MPU Control Register, MPU_CTRL* on page B3-637
0xE000ED98	MPU_RNR	RW	UNKNOWN	*MPU Region Number Register, MPU_RNR* on page B3-638
0xE000ED9C	MPU_RBAR	RW	UNKNOWN	*MPU Region Base Address Register, MPU_RBAR* on page B3-639
0xE000EDA0	MPU_RASR	RW	UNKNOWN	*MPU Region Attribute and Size Register, MPU_RASR* on page B3-640
0xE000EDA4	MPU_RBAR_A1	RW	-	Alias 1 of MPU_RBAR, see *MPU alias register support* on page B3-642
0xE000EDA8	MPU_RASR_A1	RW	-	Alias 1 of MPU_RASR, see *MPU alias register support* on page B3-642

Table B3-11 MPU register summary (continued)

Address	Name	Type	Reset	Description
0xE000EDAC	MPU_RBAR_A2	RW	-	Alias 2 of MPU_RBAR, see *MPU alias register support* on page B3-642
0xE000EDB0	MPU_RASR_A2	RW	-	Alias 2 of MPU_RASR, see *MPU alias register support* on page B3-642
0xE000EDB4	MPU_RBAR_A3	RW	-	Alias 3 of MPU_RBAR, see *MPU alias register support* on page B3-642
0xE000EDB8	MPU_RASR_A3	RW	-	Alias 3 of MPU_RASR, see *MPU alias register support* on page B3-642
0xE000EDBC-0xE000EDEC	-	…	-	Reserved.

Note

The values of the MPU_RASR registers from reset are UNKNOWN. All MPU_RASR registers must be programmed as either enabled or disabled, before enabling the MPU using the MPU_CTRL register.

B3.5.5 MPU Type Register, MPU_TYPE

The MPU_TYPE register characteristics are:

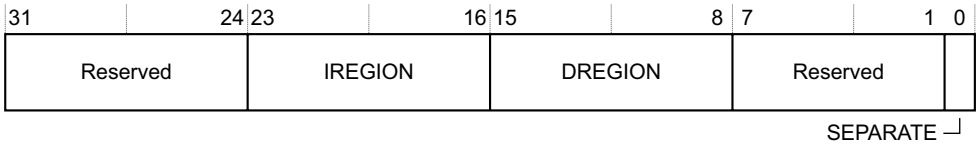
Purpose The MPU Type Register indicates how many regions the MPU support. Software can use it to determine if the processor implements an MPU.

Usage constraints There are no usage constraints.

Configurations Always implemented.

Attributes See Table B3-11 on page B3-635.

The MPU_TYPE register bit assignments are:



Bits[31:24] Reserved.

IREGION, bits[23:16]

Instruction region. RAZ. Armv7-M only supports a unified MPU.

DREGION, bits[15:8]

Number of regions supported by the MPU. If this field reads-as-zero the processor does not implement an MPU.

Bits[7:1] Reserved.

SEPARATE, bit[0] Indicates support for separate instruction and data address maps. RAZ. Armv7-M only supports a unified MPU.

B3.5.6 MPU Control Register, MPU_CTRL

The MPU_CTRL Register characteristics are:

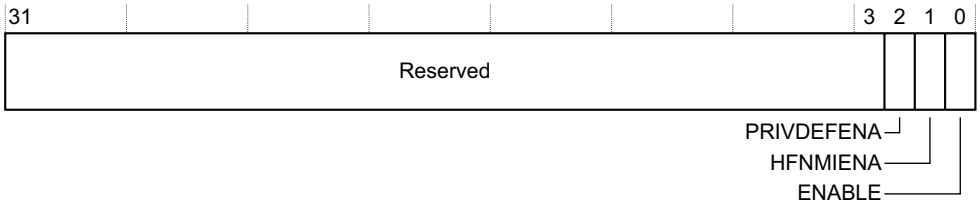
Purpose Enables the MPU, and when the MPU is enabled, controls whether the default memory map is enabled as a background region for privileged accesses, and whether the MPU is enabled for HardFaults, NMIs, and exception handlers when FAULTMASK is set to 1.

Usage constraints There are no usage constraints.

Configurations If the MPU is not implemented, this register is RAZ/WI.

Attributes See Table B3-11 on page B3-635.

The MPU_CTRL bit assignments are:



Bits[31:3] Reserved.

PRIVDEFENA, bit[2]

When the ENABLE bit is set to 1, the meaning of this bit is:

0 Disables the default memory map. Any instruction or data access that does not access a defined region faults.

1 Enables the default memory map as a background region for privileged access. The background region acts as region number -1. All memory regions configured in the MPU take priority over the default memory map. *The system address map* on page B3-592 describes the default memory map.

When the ENABLE bit is set to 0, the processor ignores the PRIVDEFENA bit.

If no regions are enabled and the PRIVDEFENA and ENABLE bits are set to 1, only privileged code can execute from the system address map.

HFNMIENA, bit[1] When the ENABLE bit is set to 1, controls whether handlers executing with priority less than 0 access memory with the MPU enabled or with the MPU disabled. This applies to HardFaults, NMIs, and exception handlers when FAULTMASK is set to 1:

0 Disables the MPU for these handlers.

1 Use the MPU for memory accesses by these handlers.

If HFNMIENA is set to 1 when ENABLE is set to 0, behavior is UNPREDICTABLE.

ENABLE, bit[0] Enables the MPU:

0 The MPU is disabled.

1 The MPU is enabled.

Disabling the MPU, by setting the ENABLE bit to 0, means that privileged and unprivileged accesses use the default memory map.

Effect of MPU_CTRL settings on unprivileged instructions

The Thumb instruction set includes instructions that, when executed by privileged software, perform unprivileged memory accesses:

- The following sections describe instructions that perform unprivileged register loads:
  - *LDRBT* on page A7-256.
  - *LDRHT* on page A7-269.

  - — *LDRSBT* on page A7-274.
  - — *LDRSHT* on page A7-280.
  - — *LDRT* on page A7-281.
- The following sections describe instructions that perform unprivileged register stores:
  - — *STRBT* on page A7-392.
  - — *STRHT* on page A7-400.
  - — *STRT* on page A7-401.

Table B3-12 shows how the MPU_CTRL.HFNMIENA and MPU_CTRL.ENABLE bits affect the handling of these instructions when issued by an exception handler for HardFault, or NMI, or for another exception when FAULTMASK is set to 1, and when this is different for other privileged software.

Table B3-12 Effect of MPU_CTRL settings on unprivileged instructions

MPU_CTRL		Effect on unprivileged load or store instructions from	
HFNMIENA	ENABLE	Specified handlers[a]	Other privileged software
0	0	MPU disabled. Unprivileged access, using default memory map.	
0	1	MPU disabled for these handlers. Unprivileged access, using default memory map.	Unprivileged access, using MPU.
1	0	UNPREDICTABLE. Software must not use this configuration.	
1	1	MPU enabled. Unprivileged access, using MPU.	

a. HardFault or NMI handler, or other exception handler when FAULTMASK is set to 1,

Table B3-12 shows whether the MPU configuration or the default memory map determines the attributes for the address accessed by the unprivileged load or store instruction. Handling of the instruction access then depends on those attributes. If the attributes do not permit an unprivileged access then the memory system generates a fault. If the access is from the NMI or HardFault handler, or when execution priority is -1 because FAULTMASK is set to 1, then this fault causes a lockup.

B3.5.7 MPU Region Number Register, MPU_RNR

The MPU_RNR characteristics are:

Purpose Selects the region currently accessed by MPU_RBAR and MPU_RASR.

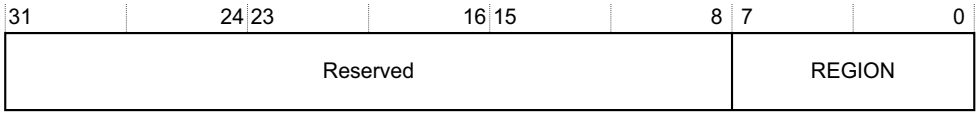
Usage constraints Used with MPU_RBAR and MPU_RASR, see *MPU Region Base Address Register, MPU_RBAR* on page B3-639, and *MPU Region Attribute and Size Register, MPU_RASR* on page B3-640.

If an implementation supports *N* regions then the regions number from 0 to (*N*-1), and the effect of writing a value of *N* or greater to the REGION field is UNPREDICTABLE.

Configurations Implemented only if the processor implements an MPU.

Attributes See Table B3-11 on page B3-635.

The MPU_RNR bit assignments are:



Bits[31:8] Reserved.

REGION, bits[7:0] Indicates the memory region accessed by MPU_RBAR and MPU_RASR.

Normally, software must write the required region number to MPU_RNR to select the required memory region, before accessing MPU_RBAR or MPU_RASR. However, the MPU_RBAR.VALID bit gives an alternative way of writing to MPU_RBAR to update a region base address without first writing the region number to MPU_RNR, see *MPU Region Base Address Register, MPU_RBAR*.

B3.5.8 MPU Region Base Address Register, MPU_RBAR

The MPU_RBAR characteristics are:

Purpose Holds the base address of the region identified by MPU_RNR. On a write, can also be used to update the base address of a specified region, in the range 0-5, updating MPU_RNR with the new region number.

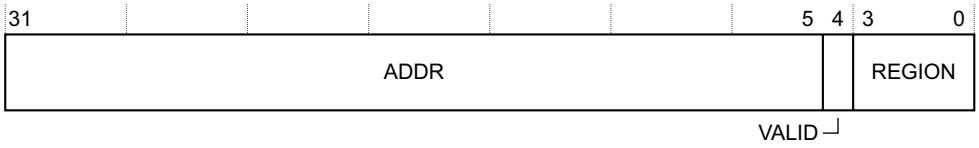
Usage constraints

- Normally, used with MPU_RBAR, see *MPU Region Number Register, MPU_RNR* on page B3-638.
- The minimum region alignment required by an MPU_RBAR is IMPLEMENTATION DEFINED. See the register description for more information about permitted region sizes.
- If an implementation supports *N* regions then the regions number from 0 to (*N*-1). If *N* is less than 16 the effect of writing a value of *N* or greater to the REGION field is UNPREDICTABLE.

Configurations Implemented only if the processor implements an MPU.

Attributes See Table B3-11 on page B3-635.

The MPU_RBAR bit assignments are:



ADDR, bits[31:5] Base address of the region.

VALID, bit[4] On writes, indicates whether the region to update is specified by MPU_RNR.REGION, or by the REGION value specified in this write. When using the REGION value specified by this write, MPU_RNR.REGION is updated to this value.

0 Apply the base address update to the region specified by MPU_RNR.REGION. The REGION field value is ignored.

1 Update MPU_RNR.REGION to the value obtained by zero extending the REGION value specified in this write, and apply the base address update to this region.

This bit reads as zero.

REGION, bits[3:0] On writes, can specify the number of the region to update, see VALID field description.

On reads, returns bits[3:0] of MPU_RNR.

Software can find the minimum size of region supported by an MPU region by writing all ones to MPU_RBAR[31:5] for that region, and then reading the register to find the value saved to bits[31:5]. The number of trailing zeros in this bit field indicates the minimum supported alignment and therefore the supported region size. An implementation must support all region size values from the minimum supported to 4GB, see the description of the MPU_RASR.SIZE field in *MPU Region Attribute and Size Register, MPU_RASR* on page B3-640.

Software must ensure that the value written to the ADDR field aligns with the size of the selected region.

B3.5.9 MPU Region Attribute and Size Register, MPU_RASR

The MPU_RASR characteristics are:

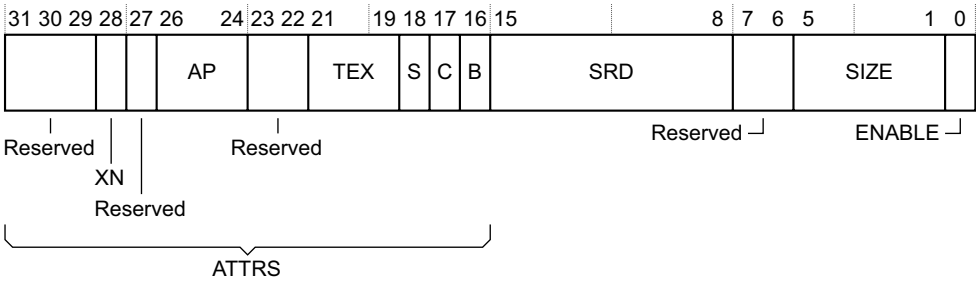
Purpose Defines the size and access behavior of the region identified by MPU_RNR, and enables that region.

Usage constraints
- Used with MPU_RNR, see *MPU Region Number Register, MPU_RNR* on page B3-638.
- Writing a SIZE value less than the minimum size supported by the corresponding MPU_RBAR has an UNPREDICTABLE effect.

Configurations Implemented only if the processor implements an MPU.

Attributes See Table B3-11 on page B3-635.

The MPU_RASR bit assignments are:



ATTRS, bits[31:16] The MPU Region Attribute field, This field has the following subfields, defined in *Region attribute control* on page B3-641:

XN MPU_RASR[28].

AP[2:0] MPU_RASR[26:24].

TEX[2:0] MPU_RASR[21:19].

S MPU_RASR[18].

C MPU_RASR[17].

B MPU_RASR[16].

SRD, bits[15:8] Subregion Disable. For regions of 256 bytes or larger, each bit of this field controls whether one of the eight equal subregions is enabled, see *Memory region subregions*:

0 Subregion enabled.

1 Subregion disabled.

Bits[7:6] Reserved.

SIZE, bits[5:1] Indicates the region size. The region size, in bytes, is $2^{(SIZE+1)}$. SIZE field values less than 4 are reserved, because the smallest supported region size is 32 bytes.

ENABLE, bit[0] Enables this region:

0 When the MPU is enabled, this region is disabled.

1 When the MPU is enabled, this region is enabled.

Enabling a region has no effect unless the MPU_CTRL.ENABLE bit is set to 1, to enable the MPU.

Memory region subregions

For any region of 256 bytes or larger, the MPU divides the region into eight equally-sized subregions. Setting a bit in the SRD field to 1 disables the corresponding subregion:

- The least significant bit of the field, MPU_RASR[8], controls the subregion with the lowest address range.
- The most significant bit of the field, MPU_RASR[15], controls the subregion with the highest address range.

For region sizes of 32, 64, and 128 bytes, the effect of setting one or more bits of the SRD field to 1 is UNPREDICTABLE.

See *Sub-region support* on page B3-633 for more information.

Region attribute control

The MPU_RASR.ATTRS field defines the memory type, and where necessary the cacheable, shareable, and access and privilege properties of the memory region. The register diagram shows the subfields of this field, where:

- The TEX[2:0], C, and B bits together indicate the memory type of the region, and:
  - For Normal memory, the cacheable properties of the region.
  - For Device memory, whether the region is shareable.

  See Table B3-13 for the encoding of these bits.
- For Normal memory regions, the S bit indicates whether the region is shareable, see Table B3-13. For Strongly-ordered and Device memory, the S bit is ignored.
- The AP[2:0] bits indicate the access and privilege properties of the region, see Table B3-15 on page B3-642.
- The XN bit is an Execute Never bit, that indicates whether the processor can execute instructions from the region, see *Execute Never encoding* on page B3-642.

Table B3-13 TEX, C, B, and S Encoding

TEX	C	B	Memory type	Description, or Normal region Cacheability	Shareable?
000	0	0	Strongly-ordered	Strongly ordered	Shareable
000	0	1	Device	Shared device	Shareable
000	1	0	Normal	Outer and inner Write-Through, no write allocate	S bit[a]
000	1	1	Normal	Outer and inner write-back, no write allocate	S bit[a]
001	0	0	Normal	Outer and inner Non-cacheable	S bit[a]
001	0	1	Reserved	Reserved	Reserved
001	1	0	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED
001	1	1	Normal	Outer and inner write-back; write and read allocate	S bit[a]
010	0	0	Device	Non-shared device	Non-shareable
010	0	1	Reserved	Reserved	Reserved
010	1	X	Reserved	Reserved	Reserved
011	X	X	Reserved	Reserved	Reserved
1BB	A	A	Normal	Cached memory, with AA and BB indicating the inner and outer Cacheability rules that must be exported on the bus. See Table B3-14 on page B3-642 for the Cacheability policy encoding. BB = Outer policy, AA == Inner policy.	S bit[a]

a. Shareable if the S bit is set to 1, Non-shareable if the S bit is set to 0

Table B3-14 Cache policy encoding

AA or BB subfield of {TEX,C,B} encoding	Cacheability policy
00	Non-cacheable
01	Write-Back, write and read allocate
10	Write-Through, no write allocate
11	Write-Back, no write allocate

The AP bits, AP[2:0], are used for access permissions. These are shown in Table B3-15.

Table B3-15 Access permissions field encoding

AP[2:0]	Privileged access	Unprivileged access	Notes
000	No access	No access	Any access generates a permission fault
001	Read/write	No access	Privileged access only
010	Read/write	Read-only	Any unprivileged write generates a permission fault
011	Read/write	Read/write	Full access
100	UNPREDICTABLE	UNPREDICTABLE	Reserved
101	Read-only	No access	Privileged read-only
110	Read-only	Read-only	Privileged and unprivileged read-only
111	Read-only	Read-only	Privileged and unprivileged read-only

Execute Never encoding

The XN bit provides an Execute Never capability. For the processor to execute an instruction, the instruction must be in a memory region with:

- Read access, indicated by the AP bits, for the appropriate privilege level.
- The XN bit set to 0.

Otherwise, the processor generates a MemManage fault when it issues the instruction for execution. Therefore, Table B3-16 shows the encoding of the XN bit.

Table B3-16 Execute Never encoding

XN	Description
0	Execution of an instruction fetched from this region permitted
1	Execution of an instruction fetched from this region not permitted

B3.5.10 MPU alias register support

The MPU_RBAR and MPU_RASR form a pair of words in the address range 0xE000ED9C-0xE000EDA3. An Armv7-M processor implements aliases of this address range at offsets of +8 bytes, +16 bytes, and +24 bytes from the MPU_RBAR address of 0xE000ED9C, as Table B3-11 on page B3-635 shows. Using these register aliases with the MPU_RBAR.REGION field, and the MPU_RBAR.VALID field set to 1, software can use a stream of word writes to update efficiently up to four regions, provided all the regions accessed are in the range region 0 to region 15.