big picture, read looking for:

- what do you have to initialize (D-cache, I-cache, branch target buffer, anything else?)
- how to enable/disable MMU
- how to setup the simplest page table (linear array of 1MB sections)
- how to tell the hardware where to find the PT
- for each page table entry (PTE), which bits can you ignore (most), which do you have to set (not many)
- any alignment issues.

when doing from scratch, you'll need a plan for breaking down "it doesn't work". vm is tricky since you often have to do several (or more) things correctly before you get any useful result. it can be difficult to figure out which you did wrong, since all mistakes lead to the machine locking up.

Chapter B4
Virtual Memory System Architecture

b6-21: invalidate data / instruction cache
additional co-processor instructions: b3

This chapter describes the *Virtual Memory System Architecture* (VMSA) based on a *Memory Management Unit* (MMU). It contains the following sections:

B4.1 About the VMSA

Complex operating systems typically use a virtual memory system to provide separate, protected address spaces for different processes. Processes are dynamically allocated memory and other memory mapped system resources under the control of a *Memory Management Unit* (MMU). The MMU allows fine-grained control of a memory system through a set of virtual to physical address mappings and associated memory properties held within one or more structures known as *Translation Lookaside Buffers* (TLBs) within the MMU. The contents of the TLBs are managed through hardware translation lookups from a set of translation tables maintained in memory.

The process of doing a full translation table lookup is called a *translation table walk*. It is performed automatically by hardware, and has a significant cost in execution time, at least one main memory access, and often two. TLBs reduce the average cost of a memory access by caching the results of translation table walks. Implementations can have a unified TLB (von Neumann architecture) or separate Instruction and Data TLBs (Harvard architecture).

we don't want backwards compat, so XP=1

The VMSA has been significantly enhanced in ARMv6. This is referred to as VMSAv6. To prevent the need for a TLB invalidation on a context switch, each virtual to physical address mapping can be marked 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), the enabled set of virtual to physical address mappings can be altered. VMSAv6 has added definitions for different memory types (see *ARMv6 memory attributes - introduction* on page B2-8), and other attributes (see *Memory access control* on page B4-8). For backwards compatibility there is an XP control bit in the System Control Coprocessor, CP15 register 1, as defined in *Register 1: Control register* on page B4-40.

The set of memory properties associated with each TLB entry includes:

Memory access permission control

This controls whether a program has no-access, read-only access, or read/write access to the memory area. When an access is not permitted, a memory abort is signaled to the processor.

will use for user-level process

The level of access allowed can be affected by whether the program is running in User mode, or a privileged mode, and by the use of domains.

Memory region attributes

These describe properties of a memory region. Examples include device (VMSAv6), non-cacheable, write-through, and write-back.

Virtual-to-physical address mapping

An address generated by the ARM® processor is called a *virtual address*. The MMU allows this address to be mapped to a different *physical address*. This physical address identifies which main memory location is being accessed.

This can be used to manage the allocation of physical memory in many ways. For example, it can be used to allocate memory to different processes with potentially conflicting address maps, or to allow an application with a sparse address map to use a contiguous region of physical memory.

Note

don't use

Because of the *Fast Context Switch Extension* (FCSE, see Chapter B8), all references to virtual address in this chapter are made to the *modified virtual address* that it generates, except where explicitly stated otherwise. The virtual address and modified virtual address are equal when the FCSE mechanism is disabled (PID == zero).

The FCSE is only present in ARMv6 for backwards compatibility. Its use in new systems is deprecated.

System Control coprocessor registers allow high-level control of this system, such as the location of the translation tables. They are also used to provide status information about memory aborts to the ARM.

would use if pt or handler in VM. we won't do this.

The VMSA allows for 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 enables the worst case access time to code and data for real-time routines to be minimized and deterministic.

need this

When translation tables in memory are changed or a different translation table is selected (by writing to CP15 register 2), previously cached translation table walk results in the TLBs can cease to be valid. The VMSA therefore supplies operations to flush TLBs.

B4.1.1 Key changes introduced in VMSAv6

The following list summarizes the changes introduced in VMSAv6:

- Entries can be associated with an application space identifier, or marked as a global mapping. This eliminates the requirement for TLB flushes on most context switches.
- Access permissions extended to allow both privileged read only, and privileged/user read-only modes to be simultaneously supported. The use of the System (S) and ROM (R) bits to control access permission determination are only supported for backwards compatibility.
- Memory region attributes to mark pages shared by multiple processors. don't need.
- The use of Tiny pages, and the fine page table second level format is now obsolete.

B4.2 Memory access sequence

we can configure how TLB laid out! unusual.

When the ARM CPU generates a memory access, the MMU performs a lookup for a mapping for the requested modified virtual address in a TLB. From VMSAv6 this also includes the current ASID. Implementations can use either Harvard or unified TLBs. If the implementation has separate instruction and data TLBs, it uses:

- the instruction TLB for an instruction fetch
- the data TLB for all other accesses.

If no global mapping, or mapping for the currently selected ASID (VMSAv6), for the modified virtual address can be found in the appropriate TLB then a translation table walk is automatically performed by hardware.

Note

Prior to VMSAv6, all modified virtual address translations can be considered as globally mapped. From ARMv6, the modified virtual address should be considered as the 32-bit modified virtual address, plus the ASID value when a non-global address is accessed.

The FCSE mechanism described in Chapter B8 *Fast Context Switch Extension* is deprecated in ARMv6. Furthermore, concurrent use of both the FCSE and ASID results in UNPREDICTABLE behavior. Either the FCSE register must be cleared, or all memory declared as global.

implies we need to figure out PTE permissions and how to setup domain permissions as well or will fault.

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 whether access is permitted. If the access is not permitted the MMU signals a memory abort. Otherwise the access is allowed to proceed.
2. The memory region attributes are used to control:
   - the cache and write buffer
   - whether the access is cached or uncached
   - the target memory type
   - whether the target memory is shared or unshared.

   can turn off caching and write back buffering on a page/section basis
3. The physical address is used for any access to external or tightly coupled memory, and can be used to perform TAG matching for cache entries in physically tagged cache implementations.

Figure B4-1 on page B4-5 shows this for a cached system.

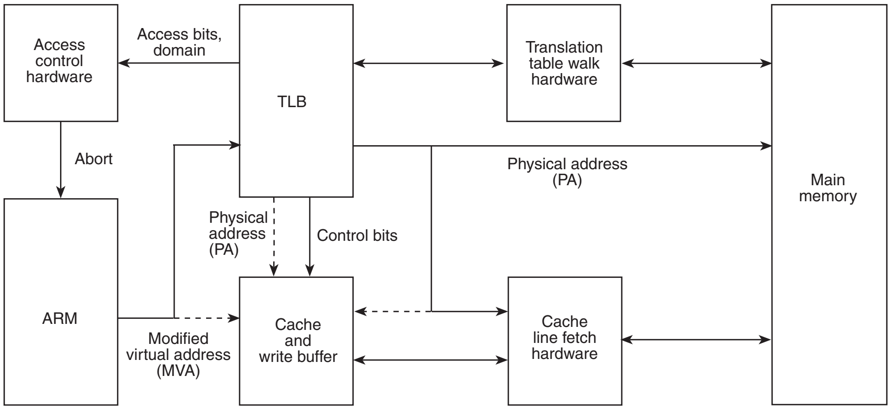


Figure B4-1 Cached MMU memory system overview

B4.2.1 TLB match process

(1) how to set,
(2) how to invalidate
when change?

Each TLB entry contains a modified virtual address, a page size, a physical address, and a set of memory properties. It is marked as being associated with a particular application space, or as global for all application spaces. Where an ASID is used, register 13 in CP15 determines the currently selected application space.

A TLB entry matches if bits 31-*N* of the modified virtual address match, and it is either marked as global, or the ASID matches the current ASID, where *N* is $\log_2$ of the page size for the TLB entry.

this is typical.

If two or more entries match at any time (including global and ASID specific entries), the behavior of a TLB is UNPREDICTABLE. The operating system must ensure that no more than one TLB entry can match at any time, typically by flushing its TLBs when global page mappings are changed.

A TLB can store entries based on the following 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.

we will use these.

Note

The use of Tiny (1KB) pages is not supported in VMSAv6.

for small processes: could simply insert a locked TLB entry and have no page table at all.

Supersections, sections and large pages are supported to allow mapping of a large region of memory while using only a single entry in a TLB.

If no mapping for an address can be 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 B4-23 for more details.

hardware TLB
replacement
so PT format
dictated by hardware
(or wouldn't be able to interpret the bits)

B4.2.2 Virtual to physical translation mapping restrictions

The VMSA can be used in conjunction with virtually-indexed, physically-tagged caches. For details of any mapping page table restrictions for virtual to physical addresses see *Restrictions on Page Table Mappings* on page B6-11.

B4.2.3 Enabling and disabling the MMU

The MMU can be enabled and disabled by writing the M bit (bit[0]) of register 1 of the System Control coprocessor. On reset, this bit is cleared to 0, disabling the MMU.

When the MMU is disabled, memory accesses are treated as follows:

if you go VM, enable caching, and code stops working, could be b/c your accesses needed to be uncached (page-table(?), devices). and/or b/c you were missing memory barriers

- All data accesses are treated as uncacheable and strongly ordered. Unexpected data cache hit behavior is IMPLEMENTATION DEFINED.

  If a Harvard cache arrangement is used then all instruction accesses are cacheable, non-sharable, normal memory if the I bit (bit[12]) of CP15 register 1 is set (1), and non-cacheable, non-sharable normal memory if the I bit is clear (0). The other cache related memory attributes (for example, Write-Through cacheable, Write-Back cacheable) are IMPLEMENTATION DEFINED.

  harvard = separate I/D

  If a unified cache is used, all instruction accesses are treated as non-shared, normal, non-cacheable.

- All explicit accesses are strongly ordered. The value of the W bit (bit[3], write buffer enable) of CP15 register 1 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 modified virtual address (this is known as a flat address mapping).

often, what "should be" is what the reset value is. be careful about "SBO" (should be 1) since easy to overwrite with 0 when setting up hardware for the first time.

- The FCSE PID (see *Register 13: Process ID on page B4-52*) *Should Be Zero* (SBZ) when the MMU is disabled. This is the reset value for the FCSE PID. If the MMU is to be disabled, the FCSE PID should be cleared. The behavior is UNPREDICTABLE if the FCSE is not cleared when the MMU is disabled.

- Cache CP15 operations act on the target cache whether the MMU is enabled or not, and regardless of the values of the memory attributes. However, if the MMU is disabled, they use the architected flat mapping.

  CP15 TLB invalidate operations act on the target TLB whether the MMU is enabled or not.

  good. otherwise would have to enable to invalidate...

- Instruction and data prefetch operations work as normal.

- Accesses to the TCMs work as normal if the TCM is enabled.

Before the MMU is enabled all relevant CP15 registers must be programmed. This includes setting up suitable translation tables in memory. Prior to enabling the MMU, the instruction cache should be disabled and invalidated. The instruction cache can then be re-enabled at the same time as the MMU is enabled.

note: if we enable caches, writebuffers then when you disable MMU you better *flush* these back, not just invalidate.

Note

Enabling or disabling the MMU effectively changes the virtual-to-physical address mapping (unless the translation tables are set up to implement a flat address mapping). Any virtually tagged caches, for example, that are enabled at the time need to be flushed (see *Memory coherency and access issues* on page B2-20).

In addition, if the physical address of the code that enables or disables the MMU differs from its modified virtual address, instruction prefetching can cause complications (see *PrefetchFlush CP15 register 7* on page B2-19). It is therefore strongly recommended that code which enables or disables the MMU has identical virtual and physical addresses.

crucial. this is one reason you'll see statements such as "the OS is mapped into every application's address space. for us, the code that enables/disables MMU will have to be in the same address in both the unmapped and VM code. various ways to do this. try to think of a clean way given our current setup.

B4.3 Memory access control

Access to a memory region is controlled by the access permission and domain bits in the TLB entry. APX and XN (execute never) bits have been added in VMSAv6. These form part of the page table entry formats described in *Hardware page table translation* on page B4-23.

B4.3.1 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, 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 register 1. For page table formats not supporting the APX bit, the value 0 is used.

Note

The use of the S and R bits is deprecated in VMSAv6. Changes to the S and R bits do not affect the access permissions of entries already in the TLB. The TLB must be flushed for the updated S and R bit values to take effect.

If an access is made to an area of memory without the required permission, a Permission Fault is raised (see *Aborts* on page B4-14).

Table Table B4-1 shows the encoding of the access permissions.

Table B4-1 MMU access permissions

S	R	APX[a]	AP[1:0]	Privileged permissions	User permissions	Description
0	0	0	0b00	No access	No access	All accesses generate permission faults
x	x	0	0b01	Read/write	No access	Privileged access only
x	x	0	0b10	Read/write	Read only	Writes in User mode generate permission faults
x	x	0	0b11	Read/write	Read/write	Full access
0	0	1	0b00	-	-	RESERVED
0	0	1	0b01	Read only	No access	Privileged read only
0	0	1	0b10	Read only	Read only	Privileged/User read only
0	0	1	0b11	-	-	RESERVED
The S and R bits are deprecated in VMSAv6. The following entries apply to legacy systems only.						
0	1	0	0b00	Read only	Read only	Privileged/User read only
1	0	0	0b00	Read only	No access	Privileged read only
1	1	0	0b00	-	-	RESERVED
0	1	1	0bxx	-	-	RESERVED
1	0	1	0bxx	-	-	RESERVED
1	1	1	0bxx	-	-	RESERVED

first step, use this

[a] VMSAv6 and above only.

we will use to detect stack corruption of return addresses

Each memory region can be tagged as not containing executable code. If the *Execute-Never* (XN) bit is set to 1, any attempt to execute an instruction in that region results in a permission fault. If the XN bit is cleared to 0, code can execute from that memory region.

Note

The XN bit acts as an additional permission check. The address must also have a valid read access.

B4.3.2 Domains

A domain is a collection of memory regions. The ARM architecture supports 16 domains. 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 two-bit field in the Domain Access Control Register. Each field allows the access to an entire domain to be enabled and disabled very quickly, so that whole memory areas can be swapped in and out of virtual memory very efficiently. Two kinds of domain access are supported:

initially make PTE use domain 0, and set domain 0 to be manager

Clients Users of domains (execute programs and access data), guarded by the access permissions of the TLB entries for that domain.

Managers Control the behavior of the domain (the current sections and pages in the domain, and the domain access), and are not guarded by the access permissions for TLB entries in that 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 allows very flexible memory protection for programs that access different memory resources. Table B4-2 shows the encoding of the bits in the Domain Access Control Register.

Table B4-2 Domain Access Values

Value	Access types	Description
0b00	No access	Any access generates a domain fault
0b01	Client	Accesses are checked against the access permission bits in the TLB entry
0b10	Reserved	Using this value has UNPREDICTABLE results
0b11	Manager	Accesses are not checked against the access permission bits in the TLB entry, so a permission fault cannot be generated

B4.4 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, and if the memory region is shareable and therefore must be kept coherent.

can control
on a PTE granularity.

Prior to VMSAv6, only C (cacheable) and B (bufferable) bits were provided. Their exact usage model (for example, how the bit settings affected write through versus write back cache policies) and any additional controls were IMPLEMENTATION DEFINED. VMSAv6 has introduced a more formal memory model (see *ARMv6 memory attributes - introduction* on page B2-8), supported by the additional bit field (TEX) and definitions described in this section.

B4.4.1 C, B, and TEX 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 B4-3 on page B4-12 shows the mapping of the *Type extension* field (TEX) and the cacheable and bufferable bits (C and B) to memory region type. For page tables formats with no TEX field the value 0b000 is used.

AFAIK for
multi-processor
we ignore. →

In addition, 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 not-shared (0). If not present, the S bit is assumed to be 0 (not-shared).

use this initially.
can experiment.
unfortunately, wrong doesn't necessarily mean code will
stop working.

Table B4-3 shows the C, B, and TEX encodings.

Table B4-3 CB + TEX Encodings

TEX	C	B	Description	Memory type	Page shareable
0b000	0	0	Strongly ordered	Strongly ordered	Shareable
0b000	0	1	Shared Device	Device	Shareable
0b000	1	0	Outer and inner write through, no write allocate	Normal	S
0b000	1	1	Outer and inner write back, no write allocate	Normal	S
0b001	0	0	Outer and inner non-cacheable	Normal	S
0b001	0	1	RESERVED	-	-
0b001	1	0	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED
0b001	1	1	Outer and inner write back, write allocate	Normal	S
0b010	0	0	Non-shared device	Device	Not shareable
0b010	0	1	RESERVED	-	-
0b010	1	X	RESERVED	-	-
0b011	X	X	RESERVED	-	-
0b1BB	A	A	Cached memory BB = outer policy, AA = inner policy	Normal	S

S indicates shareable if page table present, and S-bit in page table set, otherwise not shareable.

For an explanation of the Shareable attribute, and Normal, Strongly ordered and Device memory types see *ARMv6 memory attributes - introduction* on page B2-8.

The terms Inner and Outer refer to levels of caches that might be built in a system. Inner refers to the innermost caches, including Level 1. Outer refers to the outermost caches. The boundary between Inner and Outer caches is defined in the implementation of a cached system. Inner always includes L1. For example, in a system with three levels of caches, the Inner attributes might apply to L1 and L2, whereas the Outer attributes apply to L3. In a two-level system, it is expected that Inner applies to L1 and Outer to L2.

Table B4-4 shows the encoding of the inner and outer cache policies.

Table B4-4 Inner and outer cache policy

Encoding		Description
0	0	Non-cacheable
0	1	Write back, write allocate
1	0	Write through, no write allocate
1	1	Write back, no write allocate

It is optional which write allocation policies 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 it should not be relied on that the memory system implements that policy.

Not all inner and outer cache policies are mandatory. Table B4-5 describes the implementation options.

Table B4-5 Cache policy implementation options

Cache policy	Implementation options
Inner non-cacheable	Mandatory.
Inner write through	Mandatory.
Inner write back	Optional. If not supported, memory system should implement as inner write through.
Outer non-cacheable	Mandatory.
Outer write through	Optional. If not supported, memory system should implement as outer non-cacheable.
Outer write back	Optional. If not supported, memory system should implement as outer write through.

B4.5 Aborts

we will extend our interrupt code to handle these.

Mechanisms that can cause the ARM processor to take an exception because of a memory access are:

MMU fault The MMU detects the restriction and signals the processor.

Debug abort Monitor debug-mode 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, and use *Fault Address* and *Fault Status* registers to record associated context information. The FAR and FSR registers are described in *Fault Address and Fault Status registers* on page B4-19

B4.5.1 MMU faults

The MMU generates four types of fault:

- alignment fault
- translation fault
- domain fault
- permission fault.

start thinking about the kind of code snippet you can write to detect (1) if you are catching these and (2) how to figure out where, when, what in the exception handler.

Aborts that are detected by the MMU do not make an external access to the address that the abort was detected on.

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. See *Exceptions* on page A2-16 for more information about Prefetch and Data Aborts.

Fault-checking sequence

The sequence used by the MMU to check for access faults is slightly different for Sections and Pages. Figure B4-2 on page B4-15 shows the sequence for both types of access.

we willwrite code to detect most of these (not sub-page).

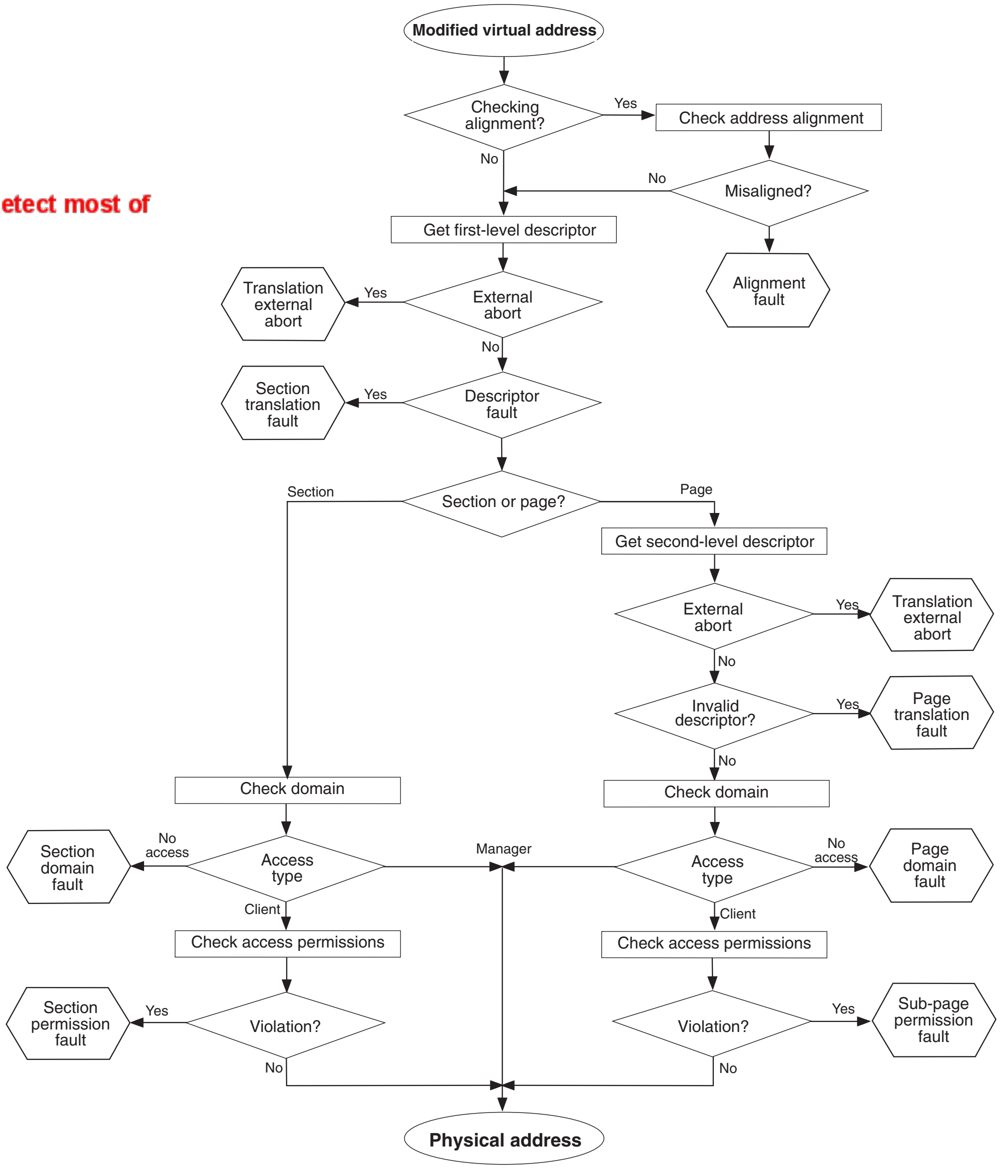


Figure B4-2 Sequence for checking faults

Alignment fault

For details of when alignment faults are generated, see Table A2-10 on page A2-40

Translation fault

There are two types of translation fault:

initialize empty PTE entries to this value.

Section This is generated if the first-level descriptor is marked as invalid. It happens when bits[1:0] of the descriptor are both 0, and in VMSAv6 formats when the value is 0b11, a RESERVED value.

Page This is generated if the second-level descriptor is marked as invalid. It happens if bits[1:0] of the descriptor are both 0.

Page Table Entry (PTE) fetches which result in translation faults are guaranteed not to be cached (no TLB updates). TLB maintenance operations are not required to flush corrupted entries on a translation fault.

again, we are only going to do sections initially

Domain fault

There are two types of domain fault:

Section domain faults

the domain is checked when the first-level descriptor is returned.

Page domain faults

the domain is checked (based on the domain field of the first level descriptor) if a valid second-level descriptor is returned.

?

Where a Domain fault results in an update to the associated page tables, it is necessary to flush the appropriate TLB entry to ensure correctness. See the page table entry update example in *TLB maintenance operations and the memory order model* on page B2-22 for more details.

Changes to the Domain Access Control register are synchronized by performing a PrefetchFlush operation (or as result of an exception or exception return). See *Changes to CP15 registers and the memory order model* on page B2-24 for details.

Permission fault

If the two-bit domain field returns *client* (01), the permission access check is performed on the access permission field in the TLB entry.

Where a permission fault results in an update to the associated page tables, it is necessary to flush the appropriate TLB entry to ensure correctness. See the page table entry update example in *TLB maintenance operations and the memory order model* on page B2-22 for more details.

when you modify the page table, the TLB won't see it. you have to manually flush TLB (either entirely, or affected entries) so that it will re-load the new, modified PTEs. this requirement is the norm.

B4.5.2 Debug events

neat: fast data breakpoints for debugging.

When Monitor debug-mode is enabled, an abort can be taken because of a breakpoint on an instruction access or a watchpoint on a data access.

If an abort is taken because of Monitor debug-mode then the appropriate FSR (instruction or data) is updated to indicate a Debug abort. This is the only information saved on a Prefetch Abort (a breakpoint) debug event. This is a precise abort. R14_abt is used to determine the address of the failing instruction.

Watchpoints are not taken precisely, because following instructions can run underneath load and store multiples. The debugger must read the *Watchpoint Fault Address Register* (WFAR) to determine which instruction caused the debug event.

B4.5.3 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 rare and are likely to be fatal to the running process. An example of an event that could cause an external memory error is an uncorrectable parity or ECC failure on a Level 2 Memory structure.

It is IMPLEMENTATION DEFINED which, if any, external aborts are supported.

The presence of a precise external abort is signaled in the DFSR or IFSR. For further details of the imprecise external abort model see *Imprecise data aborts* on page A2-23.

External abort on instruction fetch

Externally generated errors during an instruction prefetch are precise in nature, and are only recognized by the CPU if it attempts to execute the instruction fetched from the location that caused the error.

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 is not guaranteed to hold an address that is related to the instruction that caused the exception. Correspondingly, external aborts can be unrecoverable.

If an imprecise external abort causes entry into the abort state while the abort state is not re-entrant, the processor is in an unrecoverable state, as the R14 and SPSR values have been corrupted. For this reason, the existence of an imprecise external abort must only be recognized by the processor at a point when the abort state is re-entrant. This is managed by the provision of a mask for imprecise external aborts in the CSPR, which is referred to as the A bit.

Entry into the abort state caused by an imprecise external abort causes the DFSR to indicate the presence of an imprecise external abort. The FAR 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 FAR is updated on an external abort on a hardware page table walk on a data access, but not on an instruction access. The appropriate FSR (instruction or data) indicates that this has occurred.

Parity error reporting

Parity errors can occur as a precise (for example, from an L1 cache hit read) or an imprecise (for example, a cache linefill) abort. A fault status code is defined for reporting parity errors. It is IMPLEMENTATION DEFINED what parity error support is provided and whether the assigned fault status code or another appropriate encoding is used to report them.

B4.6 Fault Address and Fault Status registers

Prior to VMSAv6, the architecture supported a single *Fault Address Register* (FAR) and *Fault Status Register* (FSR).

we will use these. so figure out how to read them (see table on next page).

VMSAv6 requires four registers:

- *Instruction Fault Status Register* (IFSR) updated on Prefetch Aborts
- *Data Fault Status Register* (DFSR) updated on Data Aborts
- *Fault Address Register* (FAR) updated with the faulting address for precise exceptions
- *Watchpoint Fault Address Register* (WFAR) updated on a watchpoint access with the address of the instruction that caused the Data Abort.

Note

The IFSR and DFSR are updated on Data Aborts because of instruction cache maintenance operations.

For a description of precise and imprecise exceptions see *Exceptions* on page A2-16.

VMSAv6 added a fifth fault status bit (bit[10]) to both the IFSR and DFSR. It is IMPLEMENTATION DEFINED how this bit is encoded in earlier versions of the architecture. A write flag (bit[11] of the DFSR) has also been introduced.

we won't do paging, but we will use page protections to intercept accesses to heap memory so we can build a memory corruption checker and a race detector.

Precise aborts resulting from data accesses (Precise Data Aborts) are immediately acted upon by the CPU. The DFSR is updated with a five-bit Fault Status (FS[10,3:0]) and the domain number of the access. In addition, the modified virtual address which caused the Data Abort is written into the FAR. If a data access simultaneously generates more than one type of Data Abort, they are prioritized in the order given in Table B4-1 on page B4-20. The highest priority abort is reported.

Aborts arising from instruction fetches are flagged as the instruction enters the instruction pipeline. Only when, and if, the instruction is executed does it cause a Prefetch Abort exception. An abort resulting from an instruction fetch is not acted upon if the instruction is not used (for example, if it is branched around).

The fault address associated with a Prefetch Abort exception is determined from the value saved in R14_abt when the Prefetch Abort exception vector is entered. If the *Instruction Fault Address Register* (IFAR) is implemented, then the modified virtual address which caused the abort will also be in that register.

It is IMPLEMENTATION DEFINED whether the DFSR and FAR are updated for an abort arising from an instruction fetch, and if so, what useful information they contain about the fault. However, an abort arising from an instruction fetch never updates the DFSR and the FAR between the time that an abort arising from a data access updates them and the time of the corresponding entry into the Data Abort exception vector. In other words, a Data Abort handler can rely upon its FAR and DFSR values not being corrupted by an abort arising from an instruction fetch that was not acted upon. From VMSAv6, only the IFSR is updated by a Prefetch Abort.

Table B4-1 Fault status register encodings

Architecture	Priority	Sources		FS [10,3:0]	Domain [a]	FAR
All	Highest	Alignment		0b00001	Invalid	Valid
VMSAv6		PMSA - TLB miss (MPU)		0b00000	Invalid	Valid
		Alignment (deprecated)		0b00011		
VMSAv6		Instruction Cache Maintenance Operation Fault		0b00100	Invalid	Valid
All		External Abort on Translation	1st level	0b01100	Invalid	Valid
			2nd level	0b01110	Valid	Valid
All		Translation	Section	0b00101	Invalid	Valid
			Page	0b00111	Valid	Valid
All		Domain	Section	0b01001	Valid	Valid
			Page	0b01011	Valid	Valid
All		Permission	Section	0b01101	Valid	Valid
			Page	0b01111	Valid	Valid
VMSAv6		Precise External Abort		0b01000	Invalid	Valid
		External Abort, Precise (deprecated)		0b01010		
VMSAv6		TLB Lock [b]		0b10100	Invalid	Invalid
VMSAv6		Coprocessor Data Abort (IMPLEMENTATION DEFINED)		0b11010	Invalid	Invalid
VMSAv6		Imprecise External Abort		0b10110	Invalid	Invalid
VMSAv6		Parity Error Exception		0b11000	Invalid	IMPLEMENTATION DEFINED
VMSAv6	Lowest	Debug event		0b00010	Valid	UNPREDICTABLE

[a] domains only valid for the DFSR.

[b] see *TLB lockdown procedure - translate and lock model* on page B4-51.

B4.6.1 Notes for fault status register encodings table

Prior to VMSAv6, the usage of FS[3:0] values associated with items marked as ARMv6 is IMPLEMENTATION DEFINED. This is true for either value of FS[10].

All other FS encodings are RESERVED.

Before VMSAv6, and for VMSAv6 if the IFAR is not implemented, R14 must be used to determine the faulting address for Prefetch Aborts.

Domain information is only available for data accesses. For Prefetch Aborts, the domain information can be determined by performing a TLB lookup for the faulting address and extracting the domain field.

From VMSAv6:

- All Data Aborts cause the Data Fault Status Register (DFSR) to be updated so that the cause of the abort can be determined. All Instruction Aborts cause the Instruction Fault Status Register (IFSR) to be updated so that the cause of the abort can be determined.
- For all Data Aborts, excluding external aborts (other than on translation), the Fault Address register (FAR) will be updated with the address that caused the abort. External data aborts, other than on translation, can all be imprecise and hence the FAR does not contain the address of the abort. See section *Imprecise data aborts* on page A2-23 for more details on imprecise aborts.
- ? If a translation abort occurs during a data cache maintenance operation by modified virtual address, a Data Abort is taken and the DFSR indicates the reason. The FAR provides the faulting address.
- ? If a precise abort occurs during an instruction cache maintenance operation, then a Data Abort is taken, and an Instruction Cache Maintenance Operation Fault indicated in the DFSR. The IFSR indicates the reason. The FAR provides the faulting modified virtual address.
- The WFAR contains a copy of the PC: the address + 8 when executing in ARM state, and the address +4 when executing in Thumb® state. The value is relative to the virtual address of the instruction causing the abort, not the modified virtual address.
- The WFAR is used to store the address of the instruction that caused the watchpoint access.
- If the IFAR is implemented, it holds the faulting address for a Prefetch Abort (other than Debug aborts).

B4.6.2 Abort FSR/FAR update summary

For VMSAv6, a summary of which abort vector is taken, and which of the fault status and Fault Address registers are updated on each abort type is given in Table B4-2. The IFAR is optional.

Table B4-2 Abort FSR/FAR update summary

Abort Type	Vector	Precise	IFSR	DFSR	FAR	WFAR	IFAR
Instruction MMU fault	PABORT	Yes	Y	N	N	N	Y
Instruction debug abort	PABORT	Yes	Y	N	N	N	UNP
Instruction external abort on translation	PABORT	Yes	Y	N	N	N	Y
Instruction external abort	PABORT	Yes	Y	N	N	N	Y
Instruction cache Parity error	PABORT	Yes	Y	N	N	N	Y
Instruction cache maintenance operation	DABORT	Yes	Y	Y	Y	N	N
Data MMU fault	DABORT	Yes	N	Y	Y	N	N
Data debug abort	DABORT	No	N	Y	N	Y	N
Data external abort on translation	DABORT	Yes	N	Y	Y	N	N
Data external abort	DABORT	No	N	Y	N	N	N
Data cache Parity error	DABORT	No	N	Y	N	N	N
Data cache maintenance operation	DABORT	Yes	N	Y	Y	N	N

Here:

Y Register is updated on this abort type

N Register is not updated on this abort type.

UNP UNPREDICTABLE.

B4.7 Hardware page table translation

The MMU supports memory accesses based on sections or pages:

Supersections (optional)
Consist of 16MB blocks of memory

Sections Consist of 1MB blocks of memory.

The following page sizes are supported:

~~**Tiny pages (not in VMSAv6)**~~
~~Consist of 1KB blocks of memory.~~

ignore.

Small pages Consist of 4KB blocks of memory.

Large pages Consist of 64KB blocks of memory.

Sections and large pages are supported to allow mapping of a large region of memory while using only a single entry in the TLB. Additional access control mechanisms are extended within small pages to 1KB subpages, and within large pages to 16KB subpages. The use of subpage AP bits is deprecated in VMSAv6.

The translation table held in main memory has two levels:

we use this initially

First-level table Holds section and supersection translations, and pointers to second-level tables.

Second-level tables Hold both large and small page translations. A second form of page table, fine rather than coarse, supports tiny pages.

The MMU translates modified virtual addresses generated by the CPU into physical addresses to access external memory, and also derives and checks the access permission. Translations occur as the result of a TLB miss, and start with a first-level fetch. A section-mapped access only requires a first-level fetch, whereas a page-mapped access also requires a second-level fetch.

The value of the EE-bit in the System Control coprocessor is used to determine the endianness of the page table look ups. See *Endian configuration and control* on page A2-34 for more details.

Note

As the fine page table format and support for tiny pages is now OBSOLETE, definition of these features has been moved into a separate section, *Fine page tables and support of tiny pages* on page B4-35.

B4.7.1 Translation table base

The translation process is initiated when the on-chip TLB does not contain an entry for the requested modified virtual address. The *Translation Table Base Register* (TTBR in CP15 register 2) holds the physical address of the base of the first-level table.

easy mistake: incorrectly aligned page table.

Prior to VMSAv6, a single TTBR existed. Only bits[31:14] of the Translation Table Base Register are significant, and bits[13:0] should be zero. Therefore, the first-level page table must reside on a 16KB boundary.

VMSAv6 introduced an additional translation table base register and a translation table base control register: TTBR0, TTBR1 and TTBCR. On a TLB miss, the top bits of the modified virtual address determine if the first or second translation table base is used, see *Page table translation in VMSAv6* on page B4-25 for a detailed description of the usage model.

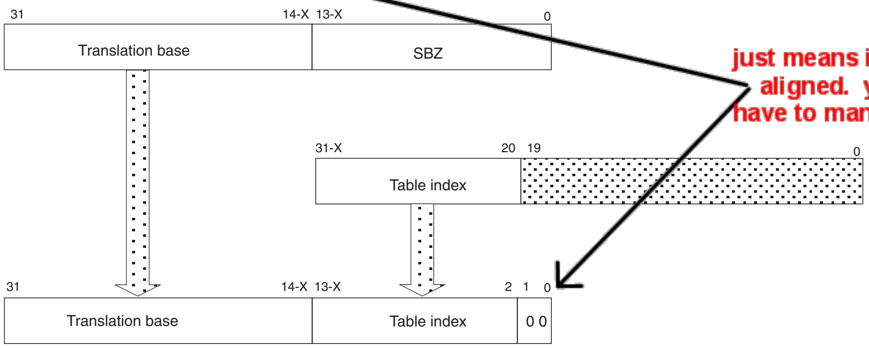
OS at high addresses, user process at lower, hole in the middle.

TTBR1 is expected to be used for operating system and I/O addresses, which do not change on a context switch. TTBR0 is expected to be used for process specific addresses. When TTBCR is programmed to zero, all translations use TTBR0 in a manner compatible with earlier versions of the architecture. The size of the TTBR1 table is always 16KB, but the TTBR0 table ranges in size from 128 bytes to 16KB, depending on the value (N) in the TTBCR, where N = 0 to 7. All translation tables must be naturally aligned.

VMSAv6 has also introduced a control bit field into the lowest bits of the TTBRs, see *Page table translation in VMSAv6* on page B4-25 for details.

B4.7.2 First-level fetch

Bits[31:14] of the Translation Table Base register are concatenated with bits[31:20] of the modified virtual address and two zero bits to produce a 32-bit physical address as shown in Figure B4-3. This address selects a four-byte translation table entry which is a first-level descriptor for a section or a pointer to a second-level page table.



just means is 4-byte aligned. you don't have to manually add.

Figure B4-3 Accessing the translation table first-level descriptors

Note

Under VMSAv6, the *Translation Base* is always address [31:14] when TTBR1 is selected. However, the value used with TTBR0 varies from address [31:14] to address [31:7] for TTBCR values of N=0 to N=7 respectively. The value of X shown in Figure B4-3 to Figure B4-7 on page B4-34 is 0 if TTBR1 is used, and is the TTBCR value N if TTBR0 is used.

Before VMSAv6, only the TTBR0 existed, and the value of X in these diagrams is always 0.

B4.7.3 Page table translation in VMSAv6

VMSAv6 supports two page table formats:

- ~~A backwards-compatible format supporting sub-page access permissions. These have been extended so certain page table entries support extended region types.~~
- A new format, not supporting sub-page access permissions, but with support for the VMSAv6 features. These features are:
  - — extended region types
  - — global and process specific pages
  - — more access permissions
  - — marking of shared and nonshared regions
  - — marking of execute-never regions.

Subpages are described in *Second-level descriptor - Coarse page table format* on page B4-31.

tricky: if you put PT in cached memory and write it, might not be in memory. a TLB replacement might not see this. i think implies PT should always be uncached.

It is IMPLEMENTATION DEFINED whether hardware page table walks can cause a read from the L1 unified/data cache. Hardware page table walks cannot cause reads from TCM. The RGN, P, S and C bits in the translation table base registers determine the memory region attributes for the page table walk. To ensure coherency on implementations that do not support page tables accesses from L1, either page tables should be stored in inner write-through memory, or if in inner write-back, the appropriate cache entries must be cleaned after modification. Page table walks may be outer-cacheable accessible as defined in the TTBR region (RGN) bits in *Translating page references in fine page tables* on page B4-38.

The page table format is selected using the XP bit in CP15 register 1. When subpage AP bits are enabled (CP15 register 1 XP = 0), the page table formats are backwards compatible with ARMv4/v5:

- all mappings are treated as global, and executable (XN = 0)
- all normal memory is nonshared
- device memory may be shared or nonshared as determined by the TEX + CB bits
- the use of subpage AP bits where AP3, AP2, AP1, AP0 contain different values is deprecated.

set XP=1

When subpage AP bits are disabled (CP15 register 1 XP = 1), the page tables have support for ARMv6 MMU features. New page table bits are added to support these features:

- The not-global (nG) bit determines whether the translation should be 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.
- The shared (S) bit, determines whether the translation is for not-shared (0), or shared (1) memory. This only applies to normal memory regions. Device memory can be shared or nonshared, as determined by the TEX + CB bits. Strongly ordered memory is always treated as shared.
- The execute-never (XN) bit determines whether 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 page table mappings support the TEX field.

Note

In VMSAv6, an invalid entry (bits[1:0] = 0b00) or a RESERVED entry (bits[1:0] = 0b11) shall result in a translation fault.

so, can initialize PTE to all 0s and get a fault if referenced.

The following sections describe the first and second level access mechanisms, and define the different table formats for VMSAv6 and earlier versions of the architecture.

B4.7.4 First-level descriptors

Each entry in the first-level table is a descriptor of how its associated 1MB modified virtual address range is mapped. Bits[1:0] of the first-level page table entry determine the type of first-level descriptor as follows:

- If bits[1:0] == 0b00, the associated modified virtual addresses are unmapped, and attempts to access them generate a translation fault (see *Aborts* on page B4-14). Software can use bits[31:2] for its own purposes in such a descriptor, as they are ignored by the hardware. Where appropriate, it is suggested that bits[31:2] continue to hold valid access permissions for the descriptor.
- If bits[1:0] == 0b10, the entry is a section descriptor for its associated modified virtual addresses. See *Sections and supersections* on page B4-28 for details of how it is interpreted.
- If bits[1:0] == 0b01, the entry gives the physical address of a coarse second-level table, that specifies how the associated 1MB modified virtual address range is mapped. Coarse tables require 1KB per table and can map large pages and small pages (see *Coarse page table descriptor* on page B4-30).
- If bits[1:0] == 0b11, the entry gives the physical address of a fine second-level table prior to VMSAv6, and is RESERVED in VMSAv6. See *Fine page tables and support of tiny pages* on page B4-35.

There are two formats of first-level descriptor table:

- VMSAv6, subpages enabled, shown in Table B4-1 on page B4-27
- VMSAv6, subpages disabled, shown in Table B4-2 on page B4-27.

The AP, APX, and domain fields are described in *Memory access control* on page B4-8. The C, B, and TEX fields are described in *Memory region attributes* on page B4-11.

The IMPLEMENTATION DEFINED (IMP) bit[9] should be set to 0 unless the implementation defined functionality enabled when bit[9]==1 is required. When this bit is 0, the implementation defined functionality is disabled.

~~**Table B4-1 First-level descriptor format (VMSAv6, subpages enabled)**~~

	31 20	19	14 12	11 10	9	8 5	4	3	2	1	0
~~Fault~~	~~IGN~~									~~0~~	~~0~~
~~Coarse page table~~	~~Coarse page table base address~~				~~IMP~~	~~Domain~~	~~SBZ~~			~~0~~	~~1~~
~~Section~~	~~Section base address~~	~~SBZ~~	~~TEX~~	~~AP~~	~~IMP~~	~~Domain~~	~~SBZ~~	~~C~~	~~B~~	~~1~~	~~0~~
	~~RESERVED~~									~~1~~	~~1~~

gives the bit offsets of each field. starts at 0, goes to 31 (4 bytes) so, e.g., IMP is a 1 bit wide field, that starts at bit offset 9. (1<<9) sets it to 1.

Table B4-2 First-level descriptor format (VMSAv6, subpages disabled)

fault

	31 24	23 20	19	18	17	16	15	14 12	11 10	9	8 5	4	3	2	1	0
Fault	IGN														0	0
Coarse page table	Coarse page table base address									IMP	Domain	SBZ			0	1
Section	Section base address		SBZ	0	nG	S	APX	TEX	AP	IMP	Domain	XN	C	B	1	0
Supersection	Supersection base address	Base address [35:32]	SBZ	1	nG	S	APX	TEX	AP	IMP	Base address [39:36]	XN	C	B	1	0
	RESERVED														1	1

careful about alignment! lower 20 bits better be 0 for the base address

is the tag to indicate is a section

make sure you figure out reasonable default values for the fields. previous page has most. shared is unclear. i think S=0 is reasonable.
SBZ = "should be zero"

we don't use. needs IMP=1 and replicated PTEs (common)

B4.7.5 Sections and supersections

If Bits[1:0] equal 0b10, the first-level descriptor is a 1MB section or a 16MB supersection descriptor.

Supersections are optional. If used, they translate 32-bit modified virtual addresses to a larger physical address space (up to eight additional address bits), and are defined as follows:

- The bit fields are described in the VMSAv6 revised format. See *First-level descriptor format (VMSAv6, subpages disabled)* on page B4-27.

Bits[1:0]	= 0b10
Bit[18]	= 0 defines a 1MB section = 1 defines a 16MB supersection
Bits[8:5]	optional extended physical address bits; PA[39:36]
Bits[23:20]	optional extended physical address bits; PA[35:32].

we don't use

- It is IMPLEMENTATION DEFINED how many additional address bits are supported.
- Supersections default to domain 0.
- It is IMPLEMENTATION DEFINED whether supersections are offered in section descriptor formats prior to ARMv6.

Figure B4-4 on page B4-29 shows how virtual to physical addresses are generated for sections. The shaded area of the descriptor represents the access control data fields.

Note

The access permissions in the first-level descriptor must be checked before the physical address is generated. The sequence for checking access permissions is given in *Access permissions* on page B4-8.

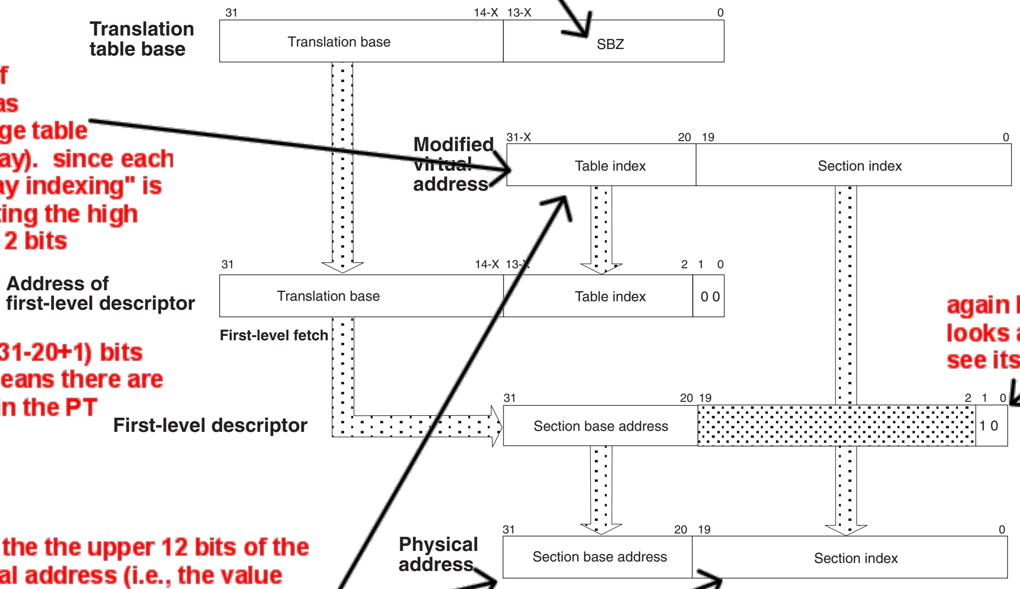


Figure B4-4 Section translation

i.e., PT base is 14 bit aligned if X=0
(a single page table)

use the upper bits of
the virtual address as
an index into the page table
(which is just an array). since each
PTE is 4 bytes, "array indexing" is
just a matter of shifting the high
bits of the VA up by 2 bits
and concatenating
them
to the PT base.
since there are 12 (31-20+1) bits
for the index, this means there are
4k possible entries in the PT

again hardware
looks at this to
see its a section

swaps the the upper 12 bits of the
physical address (i.e., the value
of base in the PTE) for the upper
12 bits of the virtual address. this
is the physical address the data
resides at.

you can see that each section is
1MB since 2^20 (the lower bits we
use to index into the section) = 1M.

B4.7.6 Coarse page table descriptor

If the first-level descriptor is a coarse page table descriptor, the fields have the following meanings:

Bits[1:0] Identify the type of descriptor (0b01 marks a coarse page table descriptor).

Bits[4:2] The meaning of these bits is IMPLEMENTATION DEFINED. From VMSAv6 these bits SBZ.

Bits[8:5] The domain field specifies one of the 16 possible domains for all the pages controlled by this descriptor.

Bit[9] IMPLEMENTATION DEFINED.

Bits[31:10] The Page Table Base Address is a pointer to a coarse second-level page table, giving the base address for a second-level fetch to be performed. Coarse second-level page tables must be aligned on a 1KB boundary.

we initially skip this. same method of ripping off upper bits, using that to index into the page table array and swapping the result back. we just do it for two levels. this gives finer control over the address protections (rather than having to use 1MB sections).

If a coarse page table descriptor is returned from the first-level fetch, a second-level fetch is initiated to retrieve a second-level descriptor, as shown in Figure B4-5. The shaded area of the descriptor represents the access control data fields.

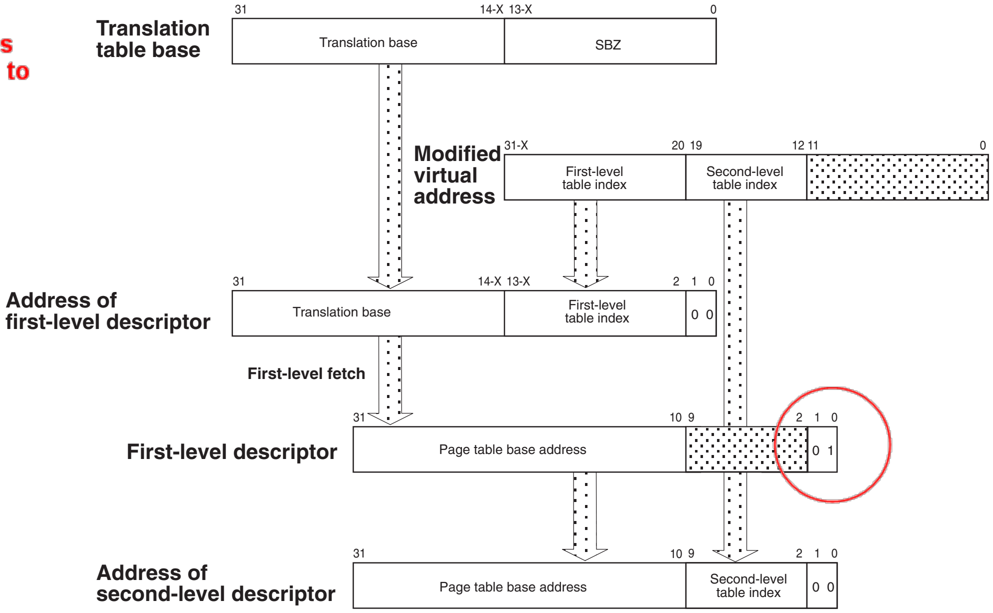


Figure B4-5 Accessing coarse page table second-level descriptors

B4.7.7 Second-level descriptor - Coarse page table format

Coarse tables are 1KB in size. Each 32-bit entry provides translation information for 4KB of memory.

VMSAv6 supports two page sizes:

- large pages are 64KB in size
- small pages are 4KB in size.

A second-level table can support page sizes greater than or equal to the amount of memory mapped by an entry. To map pages larger than the entry size, the page table entry needs to be replicated in the table the appropriate number of times:

- for a coarse table, large pages require 16 replicated entries.

There are two formats of second-level descriptor table (coarse page format):

- subpages enabled, shown in Table B4-3
- subpages disabled, shown in Table B4-4.

Table B4-3 Second-level descriptor format (subpages enabled)

	31	16	15	14	12	11	10	9	8	7	6	5	4	3	2	1	0
Fault	IGN															0	0
Large page	Large page base address		SBZ	TEX		AP3		AP2		AP1		AP0		C	B	0	1
Small page	Small page base address					AP3		AP2		AP1		AP0		C	B	1	0
Extended small page	Extended small page base address					SBZ			TEX			AP		C	B	1	1

affects alignment --- used as a way to get lower bits free for different flags

Table B4-4 Second-level descriptor format (subpages disabled)

	31	16	15	14	12	11	10	9	8	7	6	5	4	3	2	1	0
Fault	IGN															0	0
Large page	Large page base address		XN	TEX		nG	S	APX	SBZ			AP		C	B	0	1
Extended small page	Extended small page base address					nG	S	APX	TEX			AP		C	B	1	XN

Second-level page table descriptor fields

The fields in a second-level page table have the following meanings:

Bits[1:0] Identify the type of descriptor (and include XN bit in revised VMSAv6 format).

Bits[3:2] Are the cacheable and bufferable bits.

Bits[5:4] Are the access permission bits, full page or AP0 subpage.

The following bits are used for the corresponding physical address bits, the field size depending on the page size:

Bits[31:16] large (64KB) pages
Bits[31:12] small (4KB) pages

The following bits are used for additional access control functions:

same flags as
in sections

Bits[15:6] depending on the format:

- large page control
- subpage access permissions
- TEX
- APX
- S
- nG
- XN

Bits[11:6] depending on the format:

- small page control
- subpage access permissions
- TEX
- APX
- S
- nG

Bits[9:6] tiny page control, SBZ.

For details of these fields see the following sections:

AP and APX see *Access permissions* on page B4-8.

C, B and TEX see *C, B, and TEX Encodings* on page B4-11

XN, nG and S see *Page table translation in VMSAv6* on page B4-25.

Where subpages are supported, the page is divided into four blocks, each of the same size. AP0 refers to the block with the lowest block base address, with AP1, AP2 and AP3 applying to blocks with incrementing block base addresses.

B4.7.8 Translating page references in coarse page tables

Figure B4-6 shows the complete translation sequence for a 64KB large page in a coarse second-level table.

Note

Because the upper four bits of the Page Index and low-order four bits of the Second-level Table Index overlap, each page table entry for a large page must be repeated 16 times (in consecutive memory locations) in a coarse page table.

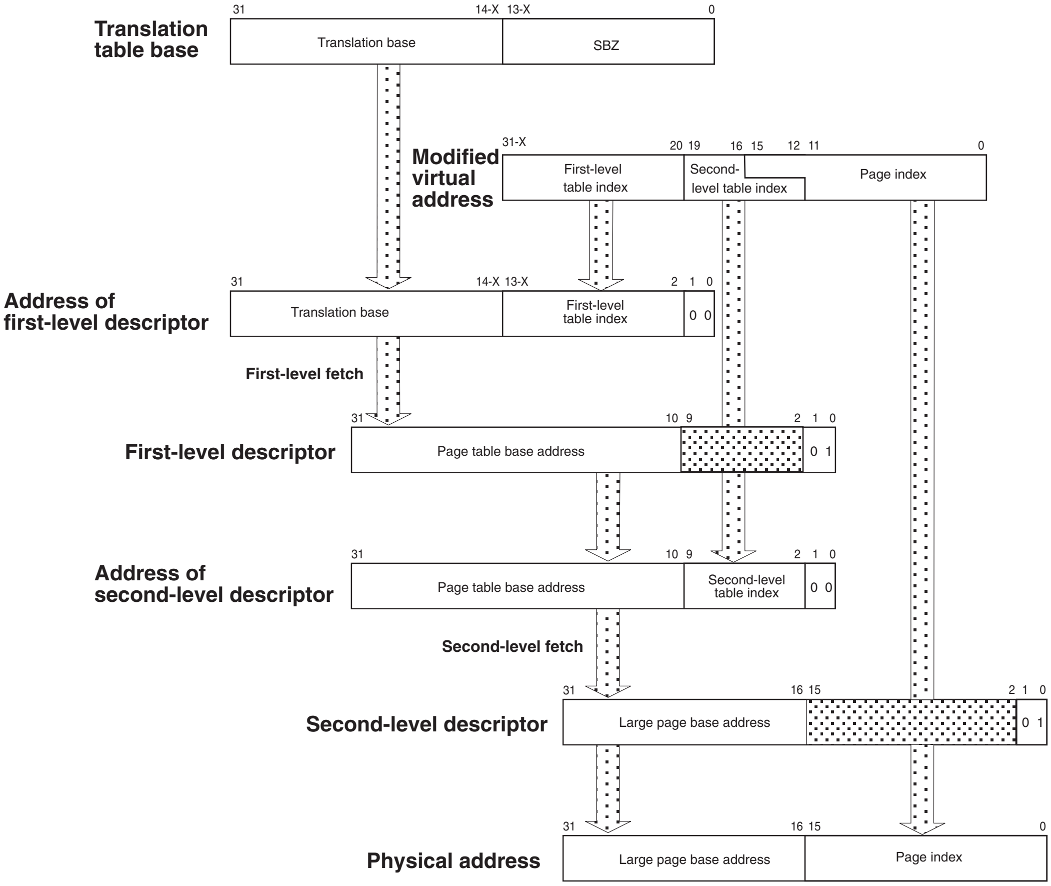


Figure B4-6 Large page translation in a coarse second-level table

Figure B4-7 shows the complete translation sequence for a 4KB small (standard or extended) page in a coarse second-level table.

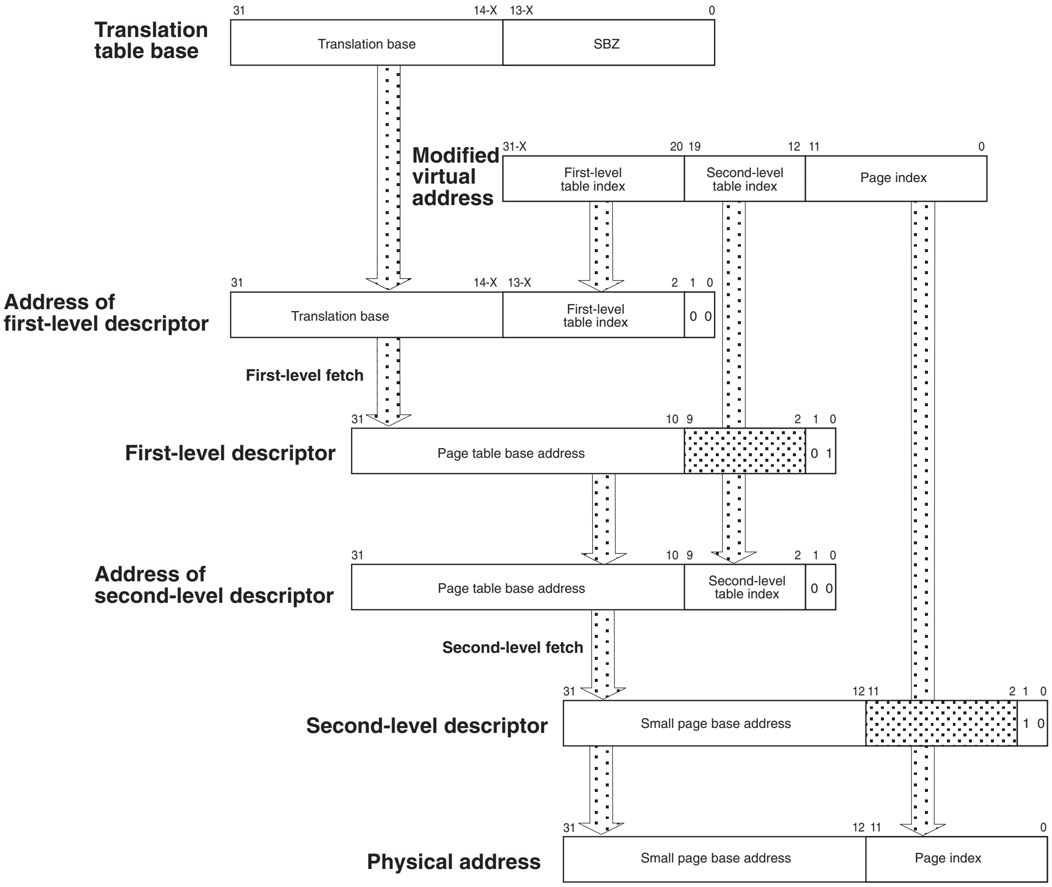


Figure B4-7 Small page translation in a coarse second-level table

B4.8 Fine page tables and support of tiny pages

Tiny pages and the fine page table format are OBSOLETE in VMSAv6. For this reason, the definition of tiny pages support and the associated first and second level descriptors is listed separately from the coarse page table formats described in *Hardware page table translation* on page B4-23.

B4.8.1 First-level descriptor

Each entry in the first-level table is a descriptor of how its associated 1MB modified virtual address range is mapped. Bits[1:0] of the first-level page table entry determine the type of first-level descriptor as follows:

- If bits[1:0] == 0b00, the associated modified virtual addresses are unmapped, and attempts to access them generate a translation fault (see *Aborts* on page B4-14). Software can use bits[31:2] for its own purposes in such a descriptor, as they are ignored by the hardware. Where appropriate, it is suggested that bits[31:2] continue to hold valid access permissions for the descriptor.
- If bits[1:0] == 0b10, the entry is a section descriptor for its associated modified virtual addresses. See *Sections and supersections* on page B4-28 for details of how it is interpreted.
- If bits[1:0] == 0b01, the entry gives the physical address of a coarse second-level table, that specifies how the associated 1MB modified virtual address range is mapped.
- If bits[1:0] == 0b11, the entry gives the physical address of a fine second-level table. A fine second-level page table specifies how the associated 1MB modified virtual address range is mapped. It requires 4KB per table, and can map large, small and tiny pages, see *Fine page tables and support of tiny pages*.

The first-level descriptor format supporting fine page tables is shown in Table B4-5.

The AP and domain fields are described in *Memory access control* on page B4-8. The C and B fields are described in *Memory region attributes* on page B4-11.

Table B4-5 First-level descriptor format

	31 … 20	19 … 14 … 12	11 10	9	8 … 5	4	3	2	1	0
Fault	IGN								0	0
Coarse page table	Coarse page table base address			SBZ	Domain	IMP			0	1
Section	Section base address	SBZ	AP	SBZ	Domain	IMP	C	B	1	0
Fine page table	Fine page table base address		SBZ		Domain	IMP			1	1

B4.8.2 Second-level descriptor

Fine tables are 4KB in size. Each 32-bit entry provides translation information for 1KB of memory.

The VMSA supports three page sizes:

- large pages are 64KB in size
- small pages are 4KB in size
- tiny pages are 1KB in size.

A second-level table can support page sizes greater than, or equal to, the amount of memory mapped by an entry. For this reason, tiny pages are only supported in fine page tables. To map pages larger than the entry size, the page table entry needs to be replicated four times for small pages and 64 times for large pages.

The second-level descriptor format supporting fine page tables is shown in Table B4-1.

Table B4-1 Second-level descriptor format

	31 … 16	15 … 12	11 10	9 8	7 6	5 4	3	2	1	0
Fault	IGN								0	0
Large page	Large table base address	SBZ	AP3	AP2	AP1	AP0	C	B	0	1
Small page	Small page base address		AP3	AP2	AP1	AP0	C	B	1	0
Tiny page	Tiny page base address			SBZ		AP	C	B	1	1

If the first-level descriptor is a fine page table descriptor, the fields have the following meanings:

Bits[1:0] Identify the type of descriptor (0b11 marks a fine page table descriptor).

Bits[4:2] The meaning of these bits is IMPLEMENTATION DEFINED.

Bits[8:5] The domain field specifies one of the sixteen possible domains for all the pages controlled by this descriptor.

Bit[11:9] These bits are not currently used, and should be zero.

Bits[31:12] The Page Table Base Address is a pointer to a fine second-level page table, giving the base address for a second-level fetch to be performed. Fine second-level page tables must be aligned on a 4KB boundary.

If a fine page table descriptor is returned from the first-level fetch, a second-level fetch is initiated to retrieve a second-level descriptor, as shown in Figure B4-8 on page B4-37. The shaded area of the descriptor represents the access control data fields.

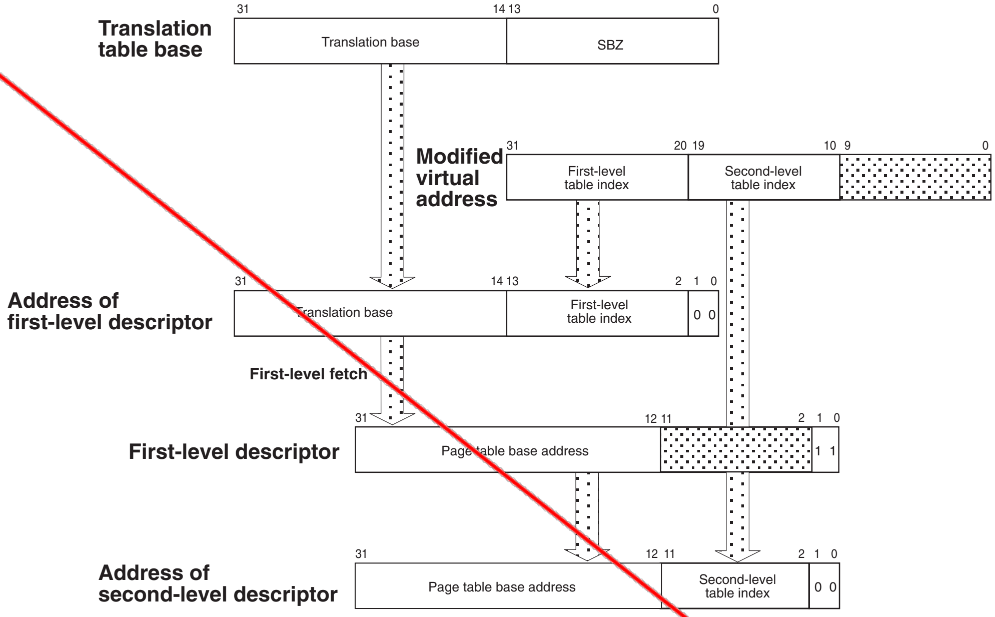


Figure B4-8 Accessing fine page table second-level descriptors

B4.8.3 Translating page references in fine page tables

The translation sequence for a large or small page in a fine second-level table is similar to that for a coarse page, but with the address of the second-level descriptor being determined as shown in Figure B4-9.

When a small page appears in a fine second-level table, the upper two bits of the Page Index and the low-order two bits of the Second-level Table Index overlap; bits[11:10]. Each page table entry for a small page must be repeated four times (in consecutive memory locations) in a fine page table. For a large page the overlap is six bits, bits[15:10], and each page entry must be repeated sixty-four times.

Tiny pages have no overlap, with one entry per 1KB page. Figure B4-9 shows the complete translation sequence for a 1KB tiny page in a fine second-level table.

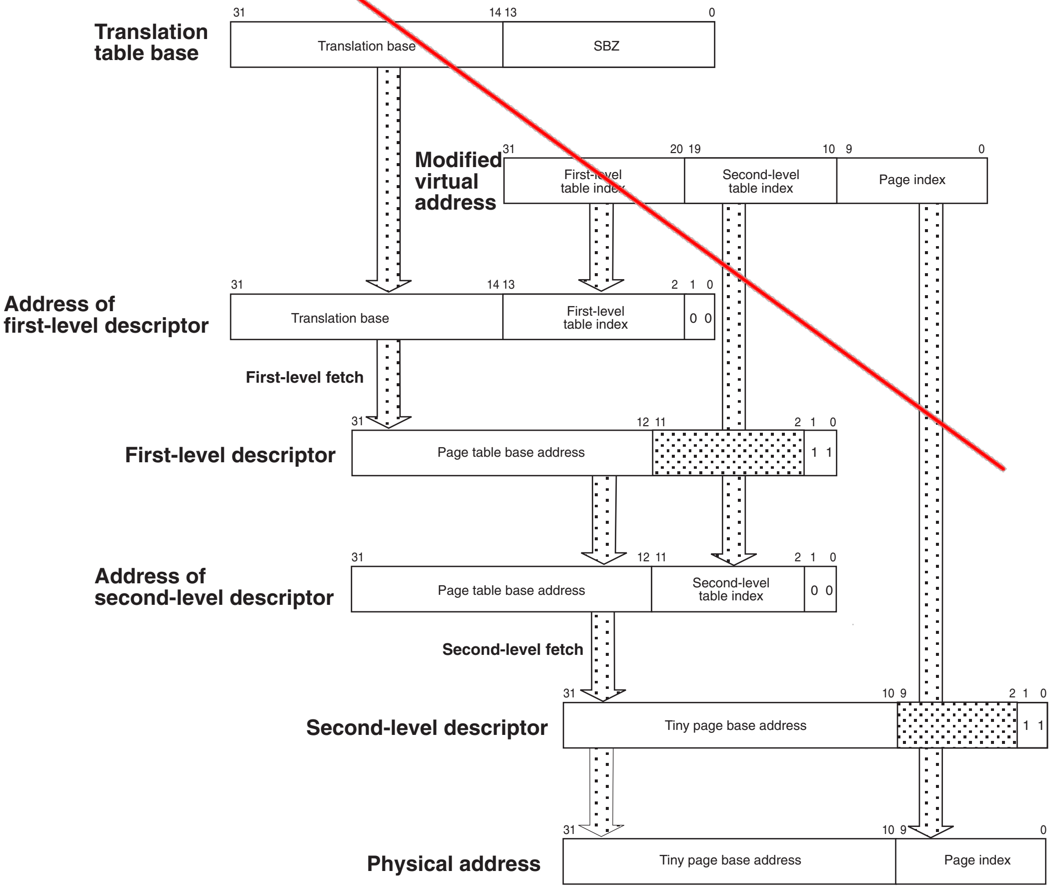


Figure B4-9 Tiny page translation in a fine second-level table

this is how you set pointer to page tables, clear TLB, clear caches, set ASID, enable/disable MMU

B4.9 CP15 registers

The MMU is controlled with the System Control coprocessor registers. From VMSAv6, several new registers, and register fields have been added:

- a TLB type register in register 0
- additional control bits to register 1
- a second translation table base register, and new control fields to register 2
- an additional fault status register to register 5
- an additional Fault Address register to register 6
- TLB invalidate by ASID support in register 8
- ASID control in register 13.

Domain support (register 3) and TLB lockdown support (register 10) are the same as in earlier versions of the architecture.

All VMSA-related registers are accessed with instructions of the form:

```
MRC p15, 0, Rd, CRn, CRm, opcode_2
MCR p15, 0, Rd, CRn, CRm, opcode_2
```

i.e., Crm = c0, opcode_2 = 0

Where CRn is the system control coprocessor register. Unless specified otherwise, CRm and opcode_2 SBZ.

B4.9.1 Register 0: TLB type register (VMSAv6)

The TLB size and organization is IMPLEMENTATION DEFINED. This read-only register describes the number of lockable TLB entries, and whether separate instruction and data or a unified TLB is present. This allows operating systems to establish how to manage the TLB. The TLB type register is accessed by reading CP15 register 0 with the opcode_2 field set to 0b011. For example:

```
MRC p15, 0, Rd, c0, c0, 3 ; returns TLB Type register
```

bit[0]	0 = Unified TLB 1 = Separate instruction/data TLBs.
Bits[7:1]	SBZ
Bits[15:8]	Number of unified/data TLB lockable entries. 0 <= N <= 255.
Bits[23:16]	Number of instruction TLB lockable entries. 0 <= N <= 255. Bits[23:16] SBZ for unified TLBs.
Bits[31:24]	SBZ

this is the main register. many more fields defined in arm1176.pdf

B4.9.2 Register 1: Control register

The following bits in the System Control coprocessor register 1 are used to control the MMU:

M (bit[0]) This is the enable/disable bit for the MMU:

0 = MMU disabled.

1 = MMU enabled.

On systems without an MMU or memory protection unit (MPU), this bit must read as zero and ignore writes.

these are the bits in the Rd register

A (bit[1]) This is the enable/disable bit for alignment fault checking (see *Alignment fault* on page B4-15):

0 = Alignment fault checking disabled

1 = Alignment fault checking enabled.

W (bit[3]) This is the enable/disable bit for the write buffer.

0 = write buffer disabled

1 = write buffer enabled.

Implementations can choose not to include the W bit. In this case this bit reads as 1 and ignores writes.

deprecated

S (bit[8]) System protection bit. This feature is deprecated from VMSAv6. The effect of this bit is defined in *Access permissions* on page B4-8.

R (bit[9]) ROM protection bit. This feature is deprecated from VMSAv6. The effect of this bit is defined in *Access permissions* on page B4-8.

XP (bit[23]) Extended page table configuration. This bit configures the hardware page table translation mechanism:

use this to disable sub-pages, legacy support. XP=1

0 = VMSAv4/v5 and VMSAv6, subpages enabled

1 = VMSAv6, subpages disabled.

EE (bit[25]) Exception Endian bit, VMSAv6 only. The EE bit is used to define the value of the CPSR E-bit on entry to an exception vector including reset. The value is also used to indicate the endianness of page table data for page table lookups. See *Endian configuration and control* on page A2-34 for more details.

B4.9.3 Register 2: Translation table base

Two translation table base registers, and a control register are provided, as shown in Table B4-2:

set register that holds page table pointer (ttbr0). can also have a second page table for the high bits in the address space. common use: put OS in this second pt and small pts in the lower ttbr0.

Table B4-2 Translation table registers

Register name	Opcode2
Translation Table Base 0 (TTBR0)	0
Translation Table Base 1(TTBR1)	1
Translation Table Base Control	2

The translation table base control register determines if a page table miss for a specific modified virtual address should use translation table base register 0, or translation table base register 1. Its format is:

31 3	2 0
UNPREDICTABLE/SBZ	N

The page table base register is selected as follows:

If N = 0 always use TTBR0. When N = 0 (the reset case), the translation table base is backwards compatible with earlier versions of the architecture.

If N > 0 then if bits [31:32-N] of the modified virtual address are all zero, use TTBR0, otherwise use TTBR1. N must be in the range 0 <= N <= 7. Therefore for N = 1; if VA[31] == 0, use TTBR0, otherwise use TTBR1. For N = 2; if VA[31:30] == 0b00 use TTBR0, otherwise use TTBR1.

next page

The format for TTBR0 is as follows:

31 14 – n	13 – n	5	4 3	2	1	0
Translation Table Base 0	UNPREDICTABLE/SBZ		RGN	IMP	S	C

Only bits [31:14-N] of the translation table base 0 register are significant. Therefore if N = 0, the page table must reside on a 16KB boundary, and, for example, if N = 1, it must reside on an 8KB boundary.

fixed.

The format for TTBR1 is as follows:

31 14	13 5	4 3	2	1	0
Translation Table Base 1	UNPREDICTABLE/SBZ	RGN	IMP	S	C

Only bits [31:14] of the translation table base 1 register are significant. Therefore TTBR1 must reside on a 16KB boundary.

The expected use for these two page table base registers is for TTBR1 to be used for operating system and I/O addresses. These do not change on context switches. TTRB0 is used for process specific addresses with each process maintaining a separate first level page table. On a context switch TTBR0 and the ContextID register are modified.

The RGN, IMP, S, and C bits provide control over the memory attributes for the page table walk:

safe: 0s for everything.

RGN Indicates if the page table memory is cacheable beyond level 1 memory:

- **00** VMSAv5: outer noncacheable
  VMSAv6: normal memory noncacheable
- **01** UNPREDICTABLE
- **10** outer cacheable write-through
- **11** outer cacheable write-back.

IMP IMPLEMENTATION DEFINED, Should-Be-Zero when not used.

S The page table walk is to shareable (1) or not-shared (0) memory.

C Page table walk is inner cacheable (1) or inner non-cacheable (0).

Note

It is IMPLEMENTATION DEFINED whether a page table walk can read from L1 cache. Therefore to ensure coherency, either page tables must be stored in inner write-through memory or, if in inner write-back, the appropriate cache entries must be cleaned after modification to ensure they are seen by the hardware page table walking mechanism.

B4.9.4 Register 3: Domain access control

safe: 0b11 for each domain.

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

The Domain Access Control register consists of 16 two-bit fields, each defining the access permissions for one of the 16 domains. Domain values are defined in *Domains* on page B4-10.

B4.9.5 Register 4: Reserved

Reading and writing CP15 register 4 is UNPREDICTABLE.

B4.9.6 Register 5: Fault status

This register enables the data and instruction fault status registers to be accessed, depending on the value of the Opcode2 field, as shown in Table B4-3. Prior to VMSAv6, only a combined FSR was defined.

Table B4-3 Fault Status Registers

Register name	Opcode2
Combined/Data FSR	0
Instruction FSR	1

Reading CP15 register 5 returns the value of the Data or Instruction Fault Status Register (DFSR/IFSR). The fault status register contains the source of the last abort. It indicates the domain (when available) and type of access being attempted when an abort occurred.

Bit[11] Added in VMSAv6. Indicates whether the aborted data access was a read (0) or write (1) access. For CP15 cache maintenance operation faults, the value read is 1. For the IFSR, this bit SBZ.

Bits[7:4] For the DFSR only, specifies which domain was being accessed when a memory system abort occurred.

Bits[10, 3:0] The reason for the abort. See Table B4-1 on page B4-20 for more information.

The IFSR and DFSR are read/write registers. This can be used to save and restore context in a debugger.

Data Fault Status Register

The format of the DFSR is as follows:

31 12	11	10	9	8	7 4	3 0
UNPREDICTABLE/SBZ	WR	FS [4]	UNP/ SBZ	0	Domain	Status

Instruction Fault Status Register

The format of the IFSR is as follows:

31 11	10	9 4	3 0
UNPREDICTABLE/SBZ	FS [4]	UNPREDICTABLE/SBZ	Status

B4.9.7 Register 6: Fault Address register

This register enables the data and watchpoint Fault Address registers to be accessed, depending on the value of the Opcode2 field, as shown in Table B4-4. Prior to VMSAv6, only a combined FAR was defined.

Table B4-4 Fault Address registers

Register name	Opcode2
Combined/Data FAR	0
Watchpoint FAR (WFAR)	1
Instruction FAR (IFAR): optional	2

The FAR, WFAR, and IFAR are updated on an abort in accordance with Table B4-2 on page B4-22.

Note

The contents of the WFAR are a virtual address, not a *Modified Virtual Address* (MVA). The FAR and IFAR contain a MVA where the FCSE mechanism is in use (see *Modified virtual addresses* on page B8-3).

The WFAR feature is migrating from CP15 to the debug architecture in CP14 and as such decoding the WFAR through CP15 is deprecated in ARMv6. See *Coprocessor 14 debug registers* on page D3-2 for its revised location.

The IFAR is optional in VMSAv6 and mandated for PMSAv6. It is only updated on prefetch aborts.

Writing CP15 register 6 enables the values of the FAR, IFAR, and WFAR to be written. This is useful for a debugger to restore their values. When the FAR is written by an `MCR` instruction, its value is treated as data, and no address modification is performed by the FCSE.

B4.9.8 Register 8: TLB functions

CP15 register 8 is a write-only register that is used to control TLBs. Table B4-5 shows the defined TLB operations and the values of `<CRm>` and `<opcode2>` used in the `MCR` instruction for each of them. The results of using any combination of `<CRm>` and `<opcode2>` not specified in the table are UNPREDICTABLE.

The synchronization of functions that update the contents of the TLB relative to their surrounding instructions is described in *TLB maintenance operations and the memory order model* on page B2-22.

The modified virtual address (MVA) is combined with the ASID for non-global pages before a translation is made. As noted in *About the FCSE* on page B8-2, the use of the FCSE with non-global pages can result in UNPREDICTABLE behavior.

Attempting to read CP15 register 8 with an `MRC` instruction is UNPREDICTABLE.

Table B4-5 TLB functions

Function	Data	Instruction
Invalidate entire unified TLB or both instruction and data TLBs	SBZ	`MCR p15, 0, Rd, c8, c7, 0`
Invalidate unified single entry	MVA	`MCR p15, 0, Rd, c8, c7, 1`
Invalidate on ASID match unified TLB	ASID	`MCR p15, 0, Rd, c8, c7, 2`
Invalidate entire instruction TLB	SBZ	`MCR p15, 0, Rd, c8, c5, 0`
Invalidate instruction single entry	MVA	`MCR p15, 0, Rd, c8, c5, 1`
Invalidate on ASID match instruction TLB	ASID	`MCR p15, 0, Rd, c8, c5, 2`
Invalidate entire data TLB	SBZ	`MCR p15, 0, Rd, c8, c6, 0`
Invalidate data single entry	MVA	`MCR p15, 0, Rd, c8, c6, 1`
Invalidate on ASID match data TLB	ASID	`MCR p15, 0, Rd, c8, c6, 2`

If the instruction or data TLB operations are used on an implementation with a unified TLB, the function is performed on the unified TLB.

Note

Since no guarantee is made that unlocked entries are held in the TLB at any point, this allows all the invalidate entire TLB operations to be treated as aliases within an implementation. A similar consideration applies for single entry operations and ASID operations in the absence of locked entries.

Invalidate TLB

This invalidates all unlocked entries in the TLB. The synchronization of the TLB maintenance operations is described in *TLB maintenance operations and the memory order model* on page B2-22.

Invalidate Single Entry

Invalidate single entry can be used to invalidate an area of memory prior to remapping. For each area of memory to be remapped (section, tiny page pre-VMSAv6, small page, or large page) an invalidate single entry of a modified virtual address in that area should be performed.

This function invalidates a TLB entry that matches the provided MVA and ASID, or a global TLB entry that matches the provided MVA. The ASID is not checked for global TLB entries for this function.

Invalidate on ASID Match

This is a single interruptible operation that invalidates all TLB entries for non-global pages which match the provided ASID.

In implementations with set-associative TLBs, this operation can take a number of cycles to complete and the instruction can be interruptible. When interrupted the R14 state is such as to indicate that the `MCR` instruction had not executed. Therefore R14 points to the address of the `MCR` + 4, The interrupt routine automatically restarts at the `MCR` instruction.

If the instruction or TLB operations are used on an implementation with a unified TLB, the equivalent function is performed on the unified TLB.

If this operation is interrupted and later restarted, it is UNPREDICTABLE whether any entries fetched into the TLB by the interrupt that use the provided ASID are invalidated by the restarted invalidation.

The Invalidate Single Entry functions require a modified virtual address as an argument. The format of the modified virtual address passed differs, depending on whether the XP control bit is set. If the XP control bit is 0, the format of the modified virtual address is simply a 32-bit MVA, and bits [11:0] are ignored. If the XP control bit is 1, the format of the modified virtual address is as follows:

31 12	11 8	7 0
MVA	IGN	ASID

Invalidate on ASID Match requires an ASID as an argument. The format is as follows:

31 8	7 0
SBZ	ASID

B4.9.9 Register 10: TLB lockdown

TLB lockdown is a feature of some ARM memory systems that allows the results of specified translation table walks to be loaded into the TLB, in such a way that they are not overwritten by the results of subsequent translation table walks. It is programmed via CP15 register 10.

we don't use.

Translation table walks can take a considerable amount of time, especially as they involve potentially slow main memory accesses. In real-time interrupt handlers, translation table walks caused by the TLB not containing translations for the handler and/or the data it accesses can increase interrupt latency significantly.

The ARM architecture supports two basic lockdown models:

- a TLB lock by entry model
- a translate and lock model.

From VMSAv6 onwards, the TLB type register can be used to discover whether a unified or Harvard TLB is implemented and the number of lockable TLB entries available. See *TLB type register* on page B3-11. In ARMv6, only the Lock by Entry model is supported. Prior to VMSAv6, any TLB locking mechanism used is IMPLEMENTATION DEFINED.

The TLB operations used to support the different mechanisms are shown in Table B4-6.

Table B4-6

Function	CRm	Opc_2	Instruction	Locking Model

Data (or unified) lockdown register	c0	0	`MCR p15,0,Rd,c10,c0,0`
			`MRC p15,0,Rd,c10,c0,0`
				Explicit

Instruction lockdown register	c0	1	`MCR p15,0,Rd,c10, c0,1`
			`MRC p15,0,Rd,c10,c0,1`
				Explicit

Translate and lock I TLB entry	c4	0	`MCR p15,0,Rd,c10,c4,0`	Trans & lock
Unlock I TLB	c4	1	`MCR p15,0,Rd,c10,c4,1`	Trans & lock
Translate and lock D TLB entry	c8	0	`MCR p15,0,Rd,c10,c4,0`	Trans & lock
Unlock D TLB	c8	1	`MCR p15,0,Rd,c10,c4,1`	Trans & lock

If W is the logarithm base 2 of the number of TLB entries, rounded up to an integer if necessary, then the format of CP15 register 10 is:

31 32-W	31-W 32-2W	31-2W 1	0
base	victim	UNP/SBZP	P

If the implementation has separate instruction and data TLBs, there are two variants of this register, selected by the `<opcode2>` field of the `MCR` or `MRC` instruction used to access register 10:

`<opcode2> == 0` Selects the data TLB lockdown register.

`<opcode2> == 1` Selects the instruction TLB lockdown register.

If the implementation has a unified TLB, only one variant of this register exists, and `<opcode2>` SBZ.

`<CRm>` must always be c0 for `MCR` and `MRC` instructions that access register 10.

Writing register 10 has the following effects:

- The victim field specifies which TLB entry is replaced by the translation table walk result generated by the next TLB miss.
- The base field constrains the TLB replacement strategy to only use the TLB entries numbered from (base) to (number of TLB entries)-1, provided the victim field is already in that range.
- Any translation table walk results written to TLB entries while P == 1 are protected from being invalidated by the register 8 invalidate entire TLB operations. Ones written while P == 0 are invalidated normally by these operations.

Note

If the number of TLB entries is not a power of two, writing a value to either the base or victim fields which is greater than or equal to the number of TLB entries has UNPREDICTABLE results.

Reading register 10 returns the last values written to the base field and the P bit, and the number of the next TLB entry to be replaced in the victim field.

The TLB lock by entry model

The incremented victim field will wrap to the value of the base field.

The architecture permits a modified form of this where the base field is fixed as zero. It is particularly appropriate where an implementation provides dedicated lockable entries (unified or Harvard) as a separate resource from the general TLB provision. To determine which form of the locking model is provided, write the base field with all bits non-zero, read it back and check whether it is a non-zero value.

The translate and lock model

This mechanism uses explicit TLB operations to translate and lock specific addresses into the TLB. Entries are unlocked on a global basis using the unlock operations. Addresses are loaded using their *Modified Virtual Address* (MVA), see *Modified virtual addresses* on page B8-3. The following actions are UNPREDICTABLE:

- accessing these functions with read (`MRC`) commands
- using functions when the MMU is disabled
- trying to translate and lock an address that is already present in the TLB.

Any Data Abort during the translation will be reported as a lock abort, see Table B4-1 on page B4-20. Only external abort or translation abort will be detected. Any access permission, domain, or alignment checks on these functions are IMPLEMENTATION DEFINED. Operations that generate an abort do not affect the target TLB.

Where this model is applied to a unified TLB, the D-side operations must be used.

Invalidate_all (`I,D` or `I and D`) operations have no effect on locked entries. Invalidate by ASID or entry is IMPLEMENTATION DEFINED with this model.

TLB lockdown procedure - by entry model

The normal procedure to lock down N TLB entries where the base field can be modified is as follows:

1. Ensure that no processor exceptions can occur during the execution of this procedure, by disabling interrupts, and so on.

2. If an instruction TLB or unified TLB is being locked down, write the appropriate version of register 10 with base == N, victim == N, and P == 0. If appropriate, also turn off facilities like branch prediction that make instruction prefetching harder to understand.

3. Invalidate the entire TLB to be locked down.

4. If an instruction TLB is being locked down, ensure that all TLB entries are loaded which relate to any instruction that could be prefetched by the rest of the lockdown procedure. (Provided care is taken about where the lockdown procedure starts, it is normally possible for one TLB entry to cover all of these, in which case the first instruction prefetch after the TLB is invalidated can do this job.)

   If a data TLB is being locked down, ensure that all TLB entries are loaded which relate to any data accessed by the rest of the lockdown procedure, including any inline literals used by its code. (This is usually best done by avoiding the use of inline literals in the lockdown procedure and by putting all other data used by it in an area covered by a single TLB entry, then loading one data item.)

   If a unified TLB is being locked down, do both of the above.

5. For each of i = 0 to N-1:

   a. Write to register 10 with base == i, victim == i, and P == 1.

   b. Force a translation table walk to occur for the area of memory whose translation table walk result is to be locked into TLB entry i, by:

      - If a data TLB or unified TLB is being locked down, loading an item of data from the area of memory.
      - If an instruction TLB is being locked down, using the register 7 *prefetch instruction cache line* operation defined in *Register 7: cache management functions* on page B6-19 to cause an instruction to be prefetched from the area of memory.

6. Write to register 10 with base == N, victim == N, and P == 0.

Note

If the *Fast Context Switch Extension* (FCSE) (see Chapter B8), is being used, care is required in step 5b, because:

- If a data TLB or a unified TLB is being locked down, the address used for the load instruction is subject to modification by the FCSE.
- If an instruction TLB is being locked down, the address used for the register 7 operation is being treated as data and so is not subject to modification by the FCSE.

To minimize the possible confusion caused by this, it is recommended that the lockdown procedure should:

- start by disabling the FCSE (by setting the PID to zero)
- where appropriate, generate modified virtual addresses itself by ORing the appropriate PID value into the top 7 bits of the virtual addresses it uses.

Where the base field is fixed at zero, the algorithm can be simplified:

1. Ensure that no processor exceptions can occur during the execution of this procedure, by disabling interrupts, and so on.

2. If any current locked entries must be removed, an appropriate sequence of invalidate single entry, or invalidate by ASID operations is required.

3. Turn off branch prediction.

4. If an instruction TLB is being locked down, ensure that all TLB entries are loaded which relate to any instruction that could be prefetched by the rest of the lockdown procedure. (Provided care is taken about where the lockdown procedure starts, it is normally possible for one TLB entry to cover all of these, in which case the first instruction prefetch after the TLB is invalidated can do this job.)

   If a data TLB is being locked down, ensure that all TLB entries are loaded which relate to any data accessed by the rest of the lockdown procedure, including any inline literals used by its code. (This is usually best done by avoiding the use of inline literals in the lockdown procedure and by putting all other data used by it in an area covered by a single TLB entry, then loading one data item.)

   If a unified TLB is being locked down, do both of the above.

5. For each of i = 0 to N-1:

   a. Write to register 10 with base == 0, victim == i, and P == 1.

   b. Force a translation table walk to occur for the area of memory whose translation table walk result is to be locked into TLB entry i, by:

      - If a data TLB or unified TLB is being locked down, loading an item of data from the area of memory.
      - If an instruction TLB is being locked down, using the register 7 *prefetch instruction cache line* operation defined in *Register 7: cache management functions* on page B6-19 to cause an instruction to be prefetched from the area of memory.

6. Clear the appropriate lockdown register.

TLB lockdown procedure - translate and lock model

All previously locked entries can be unlocked by issuing the appropriate unlock operation, I or D side. Explicit lockdown operations are then issued with the required MVA in register Rd.

TLB unlock procedure - by entry model

To unlock the locked-down portion of the TLB after it has been locked down using the above procedure:

1. Use register 8 operations to invalidate each single entry that was locked down.
2. Write to register 10 with base == 0, victim == 0, and P == 0.

Note

Step 1 is used to ensure that P == 1 entries are not left in the TLB. If they were left in the TLB, the entire TLB invalidation step (step 3) of a subsequent TLB lockdown procedure would not have the desired effect.

TLB unlock procedure - translate and lock model

Issuing the appropriate unlock (I or D) TLB operation unlocks all locked entries. It is IMPLEMENTATION DEFINED whether invalidate single entries or invalidate by ASID will remove the lock condition.

Note

The single/ASID invalidate behavior is different from the locking by entry model, where they are guaranteed to occur.

B4.9.10 Register 13: Process ID

This register determines the currently running process. Two different values can be stored depending on the opcode2 field, see Table B4-7. When updating the ASID the current instruction and data stream should be in a global, not an ASID dependent memory region. On reset, the value of the FCSE PID register *Should Be Zero* (SBZ), and the value of the ContextID register is UNDEFINED.

Table B4-7 Process ID registers

Register name	Opcode2
FCSE PID	0
ContextID	1

FCSE PID

Controls the *Fast Context Switch Extension* (FCSE). The use of the FCSE is deprecated.

31 25	24 0
~~FCSE PID~~	~~UNPREDICTABLE/SBZP~~

Context ID

The bottom eight bits of this register are the currently running ASID. The top bits extend the ASID into a general-purpose process ID.

Implementations can make this value available to the rest of the system. To ensure that all accesses are related to the correct Context ID, software should execute a Data Synchronization Barrier operation before changing this register.

The whole of this register is used by both the *Embedded Trace Macrocell* (ETM) and by the debug logic. Its value can be broadcast by the ETM to indicate the currently running process and should be programmed with a unique number for each process. Therefore if an ASID is reused the ETM can distinguish between processes. It is used by ETM to determine how virtual to physically memory is mapped.

Its value can also be used to enable process-dependent breakpoints and instructions.

The synchronization of changes to the ContextID register is discussed in *Changes to CP15 registers and the memory order model* on page B2-24.

31 25 24 8	7 0
PROCID	ASID