ARC® 700 IP Library

ARC® 700 Memory Management Unit

Reference

5127-012

ARC® 700 Memory Management Unit Reference

ARC® International

European Headquarters	North American Headquarters
ARC International,	3590 N. First Street, Suite 200
Verulam Point,	San Jose, CA 95134 USA
Station Way,	Tel. +1 408.437.3400
St Albans, Herts, AL1 5HE, UK	Fax +1 408.437.3401
Tel. +44 (0) 1727 891400	
Fax. +44 (0) 1727 891401	

www.arc.com

ARC Confidential Information

© 2004-2008 ARC International (Unpublished). All rights reserved.

Notice

This document, material and/or software contains confidential and proprietary information of ARC International and is protected by copyright, trade secret, and other state, federal, and international laws, and may be embodied in patents issued or pending. Its receipt or possession does not convey any rights to use, reproduce, disclose its contents, or to manufacture, or sell anything it may describe. Reverse engineering is prohibited, and reproduction, disclosure, or use without specific written authorization of ARC International is strictly forbidden. ARC and the ARC logotype are trademarks of ARC International.

The product described in this manual is licensed, not sold, and may be used only in accordance with the terms of a License Agreement applicable to it. Use without a License Agreement, in violation of the License Agreement, or without paying the license fee is unlawful.

Every effort is made to make this manual as accurate as possible. However, ARC International shall have no liability or responsibility to any person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this manual, including but not limited to any interruption of service, loss of business or anticipated profits, and all direct, indirect, and consequential damages resulting from the use of this manual. ARC International's entire warranty and liability in respect of use of the product are set forth in the License Agreement.

ARC International reserves the right to change the specifications and characteristics of the product described in this manual, from time to time, without notice to users. For current information on changes to the product, users should read the "readme" and/or "release notes" that are contained in the distribution media. Use of the product is subject to the warranty provisions contained in the License Agreement.

Licensee acknowledges that ARC International is the owner of all Intellectual Property rights in such documents and will ensure that an appropriate notice to that effect appears on all documents used by Licensee incorporating all or portions of this Documentation.

The manual may only be disclosed by Licensee as set forth below.

- Manuals marked "ARC Confidential & Proprietary" may be provided to Licensee's subcontractors under NDA. The manual may not be provided to any other third parties, including manufacturers. Examples--source code software, programmer guide, documentation.
- Manuals marked "ARC Confidential" may be provided to subcontractors or manufacturers for use in Licensed Products. Examples--product presentations, masks, non-RTL or non-source format.
- Manuals marked "Publicly Available" may be incorporated into Licensee's documentation with appropriate ARC permission. Examples--presentations and documentation that do not embody confidential or proprietary information.

The ARCompact instruction set architecture processor and the ARChitect configuration tool are covered by one or more of the following U.S. and international patents: U.S. Patent Nos. 6,178,547, 6,560,754, 6,718,504 and 6,848,074; Taiwan Patent Nos. 155749, 169646, and 176853; and Chinese Patent Nos. ZL 00808459.9 and 00808460.2. U.S., and international patents pending.

U.S. Government Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and/or in similar or successor clauses in the FAR, or the DOD or NASA FAR Supplement.

CONTRACTOR/MANUFACTURER IS ARC International I. P., Inc., 3590 N. First Street, Suite 200, San Jose, CA 95134.

Trademark Acknowledgments

ARCangel, ARChitect, ARCompact, ARCtangent, High C/C++, High C++, the MQX Embedded logo, RTCS, and VRaptor, are trademarks of ARC International. ARC, the ARC logo, High C, MetaWare, MQX, MQX Embedded and VTOC are registered under ARC International. All other trademarks are the property of their respective owners.

5127-012 April 2008

Contents

List of Figures

List of Tables

Chapter 1 — MMU Introduction

In this section:

Overview

The following aspects of the ARC® 700 Memory Management Unit (MMU) are covered:

- ARC 700 Memory Management Options
- ARC 700 Translation Lookaside Buffer (TLB)
- ARC 700 Page Descriptors
- Memory Mapping and Operating Modes
- Memory Management Related Exceptions
- Writing ARC 700 TLB Miss Exception Handlers

MMU Features

The MMU features are as follows:

- Software managed
  - Page Table walking, TLB entry loading
  - Marking of valid pages
  - TLB entry removal
- Hardware *suggested* replacement policy
  - The software can either rely on the hardware to supply a location for new entries, or use its own algorithm
- Unified address space for instruction and data
- Common address space for kernel and user modes
- 8-bit address space identifier (ASID)
- 4Gb physical addresses address space
- 2Gb translated memory per address space
- Fixed 8k page size
- Separate read/write/execute flags for user and kernel modes
- Cache and memory system controls
- Global access control

Memory Model

The ARC 700 processor supports virtual memory addressing if the optional Memory Management Unit (MMU) is present. If the MMU is not present or a MMU is present but is disabled, all logical addresses are mapped directly to physical addresses. By default, the MMU is disabled after reset. Note that the Data Uncached Region is always active even if the MMU is disabled.

The optional Memory Management Unit features a Translation Lookaside Buffer (TLB) for address translation and protection of 8Kb memory pages, and fixed mappings of un-translated memory. The upper half of the un-translated memory section is uncached (for IO uses) and the lower half of the un-translated memory section is cached (for operating system kernel).

The 32-bit ARC 700 architecture features a 32-bit physical address space, and a 32-bit virtual address space extended by an 8-bit address space identifier (ASID).

With the optional MMU in place, the ARC 700 architecture defines a common address space for both instruction and data accesses. The memory translation and protection systems can be arranged to provide separate non-overlapping protected regions of memory for instruction and data access within a common address space.

The ARC 700 address space is unified - separate address spaces for code and data are not permitted.

The programming interface to the Memory Management Unit has been designed to be independent of the configuration of the TLB - in terms of the associativity or number of entries.

NOTE Dirty pages are managed by an operating system using the protection bits. A ‘clean’ page will be marked as read-only. On the first write, the ‘real’ permissions will be restored and the page marked dirty. A similar scheme for reads will also be used to identify ‘used’ pages.

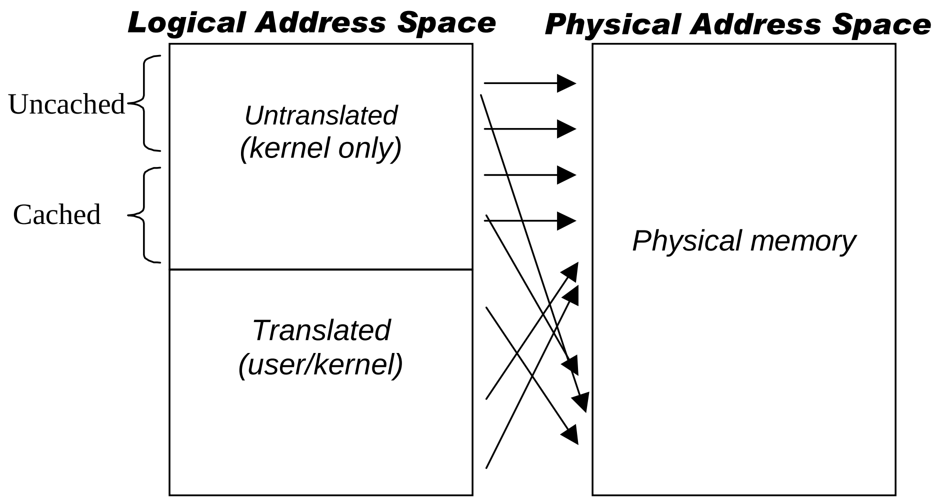


Figure 1 Architecturally Defined Address Mapping

Translation Lookaside Buffers

To provide fast translation from virtual to physical memory the MMU contains Translation Look-aside Buffers (TLBs). The MMU can be thought of as a two level cache for page descriptors: The µITLB and µDTLB at level one, and the main (or Joint) TLB at level two. The µITLB and µDTLB contain copies of the content in the Joint TLB.

In addition to providing address translation, the TLB system also provides cache control and memory protection features for individual pages.

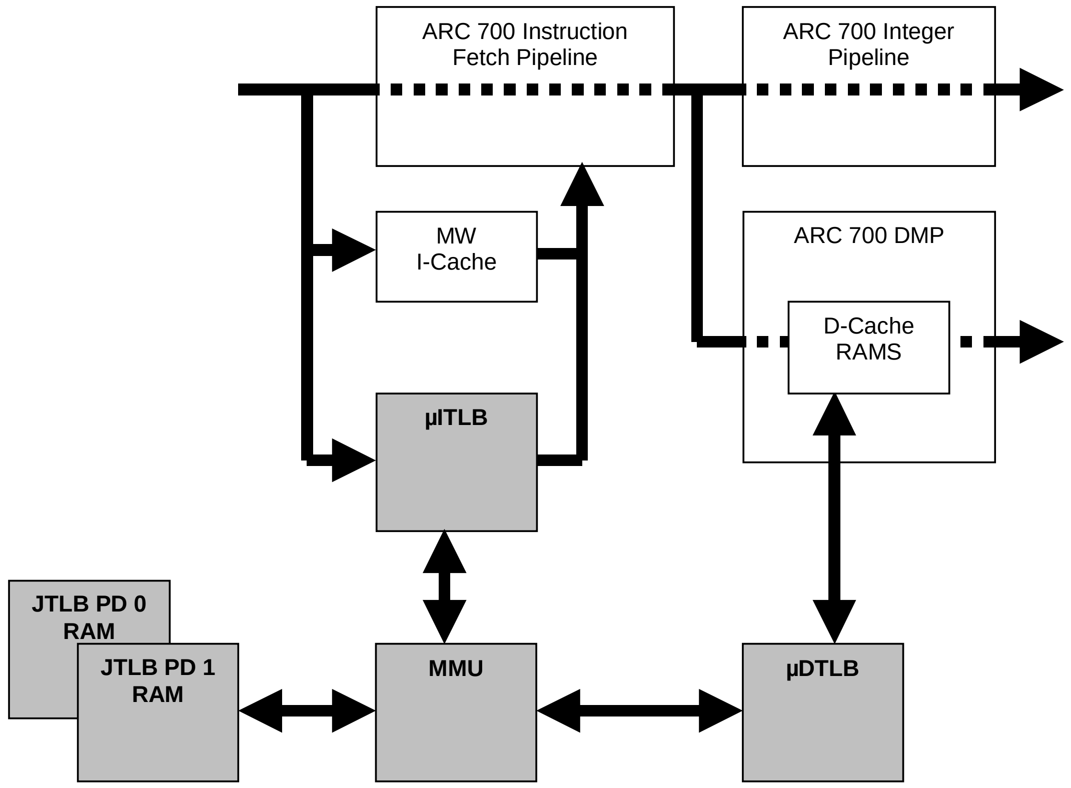


Figure 2 MMU Structure

The ARC 700 implementation features a system configured as follows:

- The µITLB and µDTLB are fully associative and physically located alongside the instruction cache and data cache, respectively, where they perform the virtual and physical address translation. The µITLB and µDTLB are hardware managed. On a µITLB (or µDTLB) page miss the hardware fetches the missing page from the main TLB.
- The Main Translation Lookaside Buffer (TLB) consists of two-way set associative Joint Translation Lookaside Buffers (JTLB), with 256 entries. The Joint TLB is software managed. On a joint TLB page miss the operating system has to fetch the missing page descriptor from memory and store it into the Joint TLB. No part of the MMU has direct access to the main memory. The Joint TLB is filled by software through an auxiliary register interface. The instruction that caused the µTLB miss is retried while the JTLB is interrogated.

Programming Model

The programming interface consists of three main components:

- Page table descriptor

- Privileged auxiliary registers for TLB access
- Memory Management Exceptions
- Physical Address Calculation

Some Memory Configuration Examples are also provided.

Chapter 2 — Page Table Descriptor

In this section:

Page Tables

Operating Systems that utilize memory management units to implement both virtual memory and address translation must maintain a data structure that describes how pages from the virtual memory space of each process relate to pages in both physical memory and external storage - such as a disk-based swap file. This data structure is called the Page Table.

The Translation Lookaside Buffer (TLB) is provided as a cache to store the most recently used entries from the Page Table. Loading of entries into the TLB from the Page Table in the ARC 700 architecture is performed under the control of software - this is generally referred to as *software page table walking*.

Since the loading of TLB entries is under the control of software, the structure of the page table is not within the scope of this information. However it is expected, but not required, that a multi-level page table structure will be implemented, with a component of the lowest level (leaf) entries being ARC 700 page descriptors, in the format described later in this section.

A second component of this structure would typically be a set of flags for the page, maintained by the OS for its own purposes.

Page Descriptor

Memory mapping is performed in blocks of 8Kb pages. The address space is unified (code and data share the same address space).

In order to map any page of physical memory into the virtual address space of a process, a page descriptor is required. This page descriptor is stored in the operating system *page table* in main memory, and the most recently used page descriptors are kept in the on-chip TLB (and in the µITLB and µDTLB) for fast access.

The page descriptor is an 8-byte structure that specifies the following for each page in use by the virtual memory system:

- In which virtual address space does it appear?
  - The ARC 700 processor allows for 256 separate virtual address spaces using an 8-bit address space identifier (ASID).
  - If a page of physical memory is to appear in more than one virtual address space, a separate page table entry is usually required for each of the address spaces in which the page appears. The only exception to this is when the page appears in all address spaces - this is a globally accessible page.
- Is it marked as *global* - available in all virtual address spaces?
- Its location in the virtual address space?
- The page in physical memory to which it is mapped
  - Or the page to be used from a swap file on disk if the page is not presently in physical memory
- The access permissions

- The ARC 700 processor allows a page to have separate read, write and execute permissions to for user and kernel mode tasks.
- This feature is provided to allow operating systems with high-reliability requirements to protect pages against unexpected accesses from both user tasks and the operating system itself.

- Whether the page table entry is valid
  - Invalid entries will not be considered by the MMU, and will be evicted before a valid entry on a TLB miss
- How the memory hierarchy should perform accesses to the page
  - Cache parameters

Restrictions of Page Mapping

The general rule is that any virtual page can be mapped to any physical page as long as it is aligned to the page size of 8Kb. However, there are three restrictions to this rule. The first one relates to shared pages, the second restriction relates to Closely Coupled Memories (CCMs) and the third relates to large cache sizes.

- Restriction for Shared Pages
- Restriction for Pages Mapped to CCMs
- Restriction for Pages Using Large Caches

Restriction for Shared Pages

When two or more virtual pages map to the same physical page, then this physical page is called a *shared page*. Shared pages can suffer from a problem called *cache aliasing*, this is when a shared physical page is held in more than one virtual address in the cache RAMs. Cache aliasing is undesirable as it is inefficient to have to check many cache RAM addresses to find a cache line.

Cache aliasing is avoided by requiring the shared page to be set to the same size as the largest cache way size in the design, but never to less than the standard page size of 8Kb. The benefit of this restriction is that the part of the virtual address which is applied to the cache RAMs is always the same as the same part of the physical address. Consequently, any virtual address can only be mapped to one single physical address.

In ARC 700 processor the cache way sizes are as follows:

- 32Kb for the 2-way 64Kb instruction cache
- 16Kb for the 2-way 32Kb instruction cache
- 16Kb for the 4-way 64Kb data cache
- 8Kb or less for all other ARC 700 cache configurations

For example, assuming the design contains a 64Kb instruction cache and a 64Kb data cache. According to the list above the cache way size is 32Kb for the instruction cache and 16Kb for the data cache. The largest of these two is the way size of the instruction cache. Consequently shared pages must be 32Kb. Larger pages are constructed by defining several contiguous 8Kb pages. In this case the following page mapping could be performed:

- Virtual page 0x10000 is mapped to physical page 0x18000 (page 1)
- Virtual page 0x12000 is mapped to physical page 0x20000 (page 2)
- Virtual page 0x14000 is mapped to physical page 0x22000 (page 3)
- Virtual page 0x16000 is mapped to physical page 0x24000 (page 4)

NOTE There are four 8Kb pages in the list above to form the shared 32Kb page. Also note that both the virtual page 0x10000 and the physical page 0x18000 are aligned to the size of the shared page (32 Kb)

Large Instruction Cache Aliasing

The ARC 700 MMU has a fixed page size of 8K bytes. Its caches are physically tagged and virtually indexed, with Instruction cache and Data cache fixed respectively at 2-way and 4-way. As cache size increase (Instruction cache beyond 16 KB and Data cache beyond 32KB), the virtual index used in cache access overlaps with the lower bits of the translated tag.

Therefore, the virtual index has the potential to no longer be guaranteed to be identical to the physical index. This ambiguity could lead to the classic cache aliasing problem. However, for large cache configurations, the cache tag field is extended and is mostly transparent to software.

Changes occur during in the low level direct RAM access by the processor. In particular, the TAG field in the IC_TAG (0x1B) auxiliary register is extended, when necessary, to hold the physical tag. In cases of large caches, this means some of the higher order bits of the INDEX field of these two registers are lost. However, the same info can be retrieved from the IC_RAM_ADDR (0x1A) auxiliary register.

Kernel code dealing with cache invalidating, cache flushing and line locking will have to deal with these differences. In particular, when writing to the registers IC_LIL (0x13) and IC_IVIL (0x19), the physical address has bits 4:0 replaced by bits 17:13 of the corresponding virtual address.

Debug operations using IC_RAM_ADDR (0x1A) are affected. When writing to IC_RAM_ADDR (0x1A) in cache controlled mode, the physical address has bits 1:0 replaced by bits 14:13 of the corresponding virtual address.

Restriction for Pages Mapped to CCMs

Pages that are mapped to either the Instruction Closely Coupled Memory (ICCM) or the Data Closely Coupled Memory (DCCM) must be of the same size as the CCM. This avoids the problem that the physical address is not available early on in the microprocessor pipeline at the time when the address is applied to the CCMs. By adhering to this restriction the part of the virtual address that is applied to the CCMs is the same as the same part of the physical address, which means that it is not necessary to wait for translation.

For example, assuming the design contains a 16Kb DCCM. Larger pages are constructed by defining several contiguous 8Kb pages. In this case the following page mapping could be performed:

- Virtual page 0x10000 is mapped to physical page 0x14000 (page 1)
- Virtual page 0x12000 is mapped to physical page 0x16000 (page 2)

NOTE There are two 8Kb-pages in the list above to form the 16Kb DCCM page. Also note that both the virtual page 0x10000 and the physical page 0x14000 are aligned to the size of the DCCM (16Kb).

Restriction for Pages Using Large Caches

The ARC 700 caches are both *virtually indexed* and *physically tagged*. This means that the address (i.e the *index*) applied to the cache RAMs is the virtual address, but the address used to compare the cache tags is the translated physical address. The micro architectural benefit of having this setup is that the cache RAM lookup can start one cycle earlier and therefore does not have to wait for the page translation to complete before accessing the cache RAMs.

For small caches the virtual and physical index are identical, but for large caches they can be different. When the virtual and physical indexes are different then one physical tag can be held in several different indexes. This problem is called *cache aliasing*. In the ARC 700 processor cache aliasing is avoided by restricting the page allocation in such a way that the virtual and physical indexes are always identical.

Cache aliasing only occurs in the ARC 700 processor for instruction caches of sizes 32KB to 64Kb and for data caches of the size 64Kb. The restrictions for page allocations are as follows:

- **64Kb data cache and 32Kb instruction cache -** address bit 13 must be the same for both the virtual and physical address. For example virtual address 0x2000 can be mapped to physical address 0x102000 but not to 0x100000.
- **64Kb instruction cache -** address bits 13-14 must be the same for both the virtual and physical address. For example virtual address 0x6000 can be mapped to physical address 0x106000 but not to 0x100000, 0x102000 or 0x104000.

Page Descriptor Format

The 32-bit ARC 700 page descriptor consists of two 32-bit words, and is arranged as follows. The first word relates to TLB Page Descriptor 0 (TLBPD0), the second to TLB Page Descriptor 1 (TLBPD1).

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
R	V[17:0]																		R		V	R	G	A[7:0]							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P[18:0]																			Reserved				RK	WK	EK	RU	WU	EU	FC	R	

The following fields are described in more detail:

- V[17:0] - Virtual Page Number
- V - Valid
- G - Global
- A[7:0] - Address Space Identifier ASID
- P[18:0] - Physical Page Number
- RK, WK, EK - Kernel Mode Permission Bits
- RU, WU, EU - User Mode Permission Bits
- FC - Cached/Uncached Flag

V[17:0] - Virtual Page Number

This 18-bit field provides the virtual page number corresponding to this page descriptor (Virtual address shifted right by 13 bits, and top bit masked off). The memory management unit checks the virtual address of an incoming request against the entries in the translation lookaside buffer. If a matching entry is found, the physical address can be calculated. The field is aligned in the page descriptor to allow generation from a 32-bit address using a simple AND operation.

Bit 31 of TLB Page Descriptor 0 is not included in this field since translated memory is only available in the lower 2Gb of the address space.

V - Valid

This bit field is used to indicate whether an entry in the TLB should be considered during a memory access when checking for a matching TLB entry. When cleared, it is used by the MMU to determine a *suggested entry* for replacement - invalid entries will always be chosen before a valid entry is evicted.

G - Global

This bit is used to indicate whether the page is *global*:

- When set true (1)
  - This page appears in all virtual address spaces
    - ASID bits for this entry are ignored, and must be set to zero
- When set false (0)
  - This page appears in a single virtual address space as described by the ASID bits A[7:0]

If a page is required to be mapped into more than one address space:

- If the page is to be available in all address spaces, with identical access requirements, a single page table entry may be used with the global bit set.
  It is the responsibility of the operating system to ensure that a globally available page does not overlap with a page at the same location in a single address space - this condition will cause a fatal machine-check exception.
  Page read/write/execute permissions are not considered when testing for multiple overlapping pages - hence it would seem to be possible to set up a global page accessible only by the operating system in kernel mode, and a page at the same address only accessible by one user mode task. However this arrangement is not permitted and will cause a fatal machine check exception.
- If the page is to be available in some but not all address spaces, or if it is to be available in all address spaces, but with different access privileges, then a separate page table entry is required for each virtual address space in use.

The global bit is typically used for mapping private operating system memory pages - in which case it would not be used with user mode read/write/execute permissions.

In other operating system or RTOS systems, this mode may be used for data or code areas shared between all processes and the operating system.

A[7:0] - Address Space Identifier ASID

These bits describe the 8-bit address space identifier (ASID) that can be considered to be an extension of the virtual address.

When the *global* bit G is set false, and a memory access is taking place, these bits are tested against the current task's ASID from the machine status register (PID) to determine whether a TLB match has taken place.

With an 8-bit ASID, it follows that the ARC 700 processor supports up to 256 concurrent virtual address spaces.

P[18:0] - Physical Page Number

This 19-bit field is used for virtual to physical address translation, as follows:

- When the entry is marked valid:
  - This virtual memory page is present in physical memory
  - This field describes the physical page in main memory that is used for accesses to the virtual page. The high order bits of the physical address (page number) come from this field, and the bottom twelve bits (page offset) come from the bottom twelve bits of the virtual address. The physical page can be located anywhere in the full 32-bit (4Gb) address space.
- When the entry is not marked valid
  - This virtual page is not present in physical memory. Only useful in an operating system supporting demand-paged virtual memory.
  - This field would typically be used by the OS to indicate the location of the page in a disk-based swap file.

The physical page number of a block is the address of the block shifted right by 13 bits (divided by 8192).

R_K, W_K, E_K - Kernel Mode Permission Bits

Each ARC 700 page descriptor features separate access control bits for user mode and kernel mode tasks.

It should be noted that these do not form part of the addressing mechanism - it is not permissible to have more than one TLB entry mapped to the same virtual address - even if the access permissions do not overlap.

An exception will be generated if an access is attempted which violates the access permissions for the page. The access will not complete.

These three bits control the permissions granted to tasks, interrupts or exception handlers running in kernel mode or other operating system functions using kernel mode. Setting the bit true (1) indicates that the permission is granted.

- R_K - Kernel mode read permission
- W_K - Kernel mode write permission
- E_K - Kernel mode execute permission

See the subsection later in this section for more discussion regarding typical setting of the permission bits.

R_U, W_U, E_U - User Mode Permission Bits

Each ARC 700 page descriptor features separate access control bits for user mode and kernel mode tasks.

It should be noted that these do not form part of the addressing mechanism - it is not permissible to have more than one TLB entry mapped to the same virtual address - even if the access permissions do not overlap.

An exception will be generated if an access is attempted which violates the access permissions for the page. The access will not complete.

These three bits control the permissions granted to tasks running in user mode. Setting the bit true (1) indicates that the permission is granted.

- R_U - User mode read permission
- W_U - User mode write permission
- E_U - User mode execute permission

See the subsection later in this section for more discussion regarding typical setting of the permission bits.

F_C - Cached/Uncached Flag

This bit controls cache operation when accessing this bit of virtual memory.

- The default condition is to set this bit true to indicate that caches may be used for accesses to this page.
- When this bit it set false, accesses to this page are sent directly to external memory, bypassing caches. This can be used for IO register space or volatile data areas - such as a region of memory where data is written by a DMA transfer.

It is the responsibility of the operating system (or user code) to ensure that the cache does not contain entries from pages that are marked *uncached*. This is to ensure that the cache does not contain old data that might either be flushed into the uncached page at some later point, or which might be used incorrectly if a page is subsequently marked *cached*.

When more that one virtual page is mapped to the same physical page, all pages must have the same setting for cached/uncached flag.

In addition to the cached/uncached switch in the page descriptor, individual load and store instructions have a cached/uncached mode switch. Accesses are performed without caches if either the instruction or the page descriptor indicates *uncached*. Access use the caches only if both the instruction and the page descriptor indicate *cached*.

TLB Indices Arrangement

The table below shows the arrangement of indices in the ARC 700 TLB:

Table 1 ARC 700 Set-Associative TLB Indices

Indices	Description
0x0	JTLB set 0 way 0
0x1	JTLB set 0 way 1
0x2	JTLB set 1 way 0
0x3	JTLB set 1 way 1
0x4	JTLB set 2 way 0
0x5	JTLB set 2 way 1
0x6	JTLB set 3 way 0
0x7	JTLB set 3 way 1
-	-
0xFC	JTLB set 127 way 0
0xFD	JTLB set 127 way 1
0xFE	JTLB set 128 way 0
0xFF	JTLB set 128 way 1
-	-
0x200	µITLB entry 0
0x201	µITLB entry 1
0x202	µITLB entry 2
0x203	µITLB entry 3
-	-
0x400	µDTLB entry 0
0x401	µDTLB entry 1
0x402	µDTLB entry 2
0x403	µDTLB entry 3
0x404	µDTLB entry 4
0x405	µDTLB entry 5
0x406	µDTLB entry 6
0x407	µDTLB entry 7

The privileged instructions for maintaining the ARC 700 TLB all use an index number. This field is also used to signal error and status information when an instruction asks for an index number to be returned.

On all ARC 700 implementations, index numbers must start from zero and be continuous. In a set-associative TLB, the *way* is specified with the low order bits of the index.

The index scheme is arranged to allow future TLBs to be implemented with different number of entries and/or different associativity. For example, a future 4-way set-associate TLB would have indexes; 0 = set 0 way 0, 1 = set 0 way 1, 2 = set 0 way 2, 3 = set 0 way 3, 4 = set 1 way 0 and so forth.

NOTE A fully associative n-entry TLB effectively has one set with n ways, hence entries would be numbered 0 = set 0 way 0, set 0 way 1 up to set 0 way n.

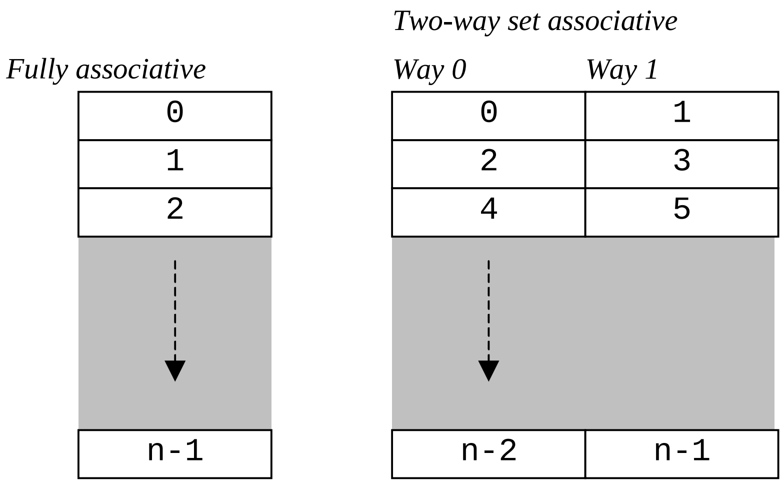
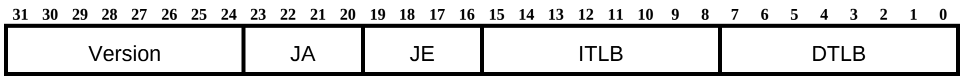


Figure 3 Fully-Associative and Set-Associative TLB Indices

MMU Build Configuration Register, MMU_BUILD

Build configuration register `MMU_BUILD (0x6F)` contains information for Operating Systems to determine the configuration of the Memory Management Unit. The default for the complete register for ARC 700 MMU version 0x1 is 0x01170408. Replacement algorithms are inferred from the version number.



The following table describes the fields in more detail.

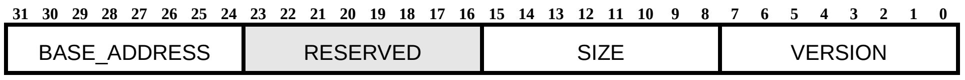
Field	Description
ITLB	*Integer number of ITLB/ µITLB entries*
	Number of µITLB entries for ARC 700 (4)
DTLB	*Integer number of DTLB/ µDTLB entries*
	Number of µDTLB entries for the ARC 700 processor (8)
JE	*Joint TLB contains 2JE entries, per way*
	ARC 700 processor defaults to 0x7 (128 entries) per way
JA	*Joint TLB contains 2JA ways*
	ARC 700 defaults to 0x1 (2 ways)
Version	*Version*
	First MMU release for the ARC 700 processor has version number 0x1

Data Uncached Build Configuration Register, DATA_UNCACHED

The build configuration register `DATA_UNCACHED (0x6A)` describes the Data Uncached region. Memory operations that access this region will always be uncached. Instruction fetches that access the same region will however be cached as this region relates to data only.

This region, which is only present in builds with an MMU, is fixed to the upper 1 Gb of the memory map. As the upper 2 Gb of the memory is the un-translated memory region, the Data Uncached region is consequently both uncached and un-translated. This makes this region suitable for e.g. peripherals. Note that this region is active even if the MMU is disabled.

The Data Uncached region is a part of the logical memory map and not part of the physical memory map. As a consequence, this region will not affect a page that is translated to a physical location that resides within the address range of the Data Uncached region. Instead, such a page would be cached or not depending on its cache flag.



The following table describes the fields in more detail.

Field	Description
Version	*Version*
	Current version number of this BCR is 0x1
SIZE	*Size of the Data Uncached region*
	0x0 - 16 MB
	0x1 - 32 MB
	0x2 - 64 MB
	0x3 - 128 MB
	0x4 - 256 MB
	0x5 - 512 MB
	0x6 - 1024 MB
	0x7 - 2048 MB
	The size is set to 0x6, i.e. 1024 MB.
BASE_ADDRESS	*Base address of the Data Uncached region*
	As it must be in the upper half of the memory space (which is the un-translated region) this means that bit 31 must always be set to 1.
	The base address is set to 0xC0.
Reserved	*Reserved*
	Should be set to zero.

Chapter 3 — Privileged Auxiliary Registers for TLB Access

In this section:

Maintenance and Control

These auxiliary registers are provided for interaction between an operating system (or debugger) and the TLB:

Table 2 Special Purpose Registers for TLB Control

Auxiliary Register	Name	Read/Write	Description
0x405	`TLBPD0`	r/w	TLB Page Descriptor register 0
0x406	`TLBPD1`	r/w	TLB Page Descriptor register 1
0x407	`TLBIndex`	r/w	TLB Index register
0x408	`TLBCommand`	w	TLB Command register (fully serializing)
0x409	`PID`	r/w	Process ID, TLB enable
0x418	`SCRATCH_DATA0`	r/w	32-bit scratch auxiliary register that can be used to store any data. The OS may for example use this register to hold the base address of the first level page table in order to speed up page table access.

These registers may only be accessed when in kernel mode. An attempt to access these registers from user mode will result in an exception.

In an ARC 700 processor write operations to the auxiliary registers are generally serializing, i.e. a pipeline flush occurs after the auxiliary write operation has committed. For best operating system performance, it is desirable to minimize time spent in TLB miss handlers - hence minimizing pipeline flushes is important.

Writes to auxiliary registers `TLBPD0`, `TLBPD1` and `TLBIndex` do not affect the operating environment of the processor until the `TLBCommand` register is written, so it is possible to make writes to auxiliary registers `TLBPD0`, `TLBPD1` and `TLBIndex` non-serializing.

Writes to the `TLBCommand` or `PID` register affects the processor operating environment directly and hence these writes are serializing. Writes to `SCRATCH_DATA0` register are not serializing. Auxiliary read operations are not serializing.

TLB Page Descriptor Registers, TLBPD0 and TLBPD1

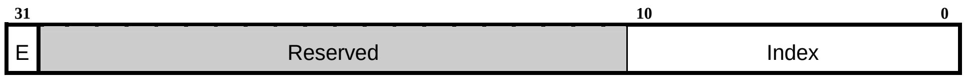
These registers are used for the following purposes:

- To supply a page descriptor for subsequent loading into the TLB
- To return a page descriptor from a TLB probe operation
- The virtual page number field is used to specify the virtual address of a TLB entry to be removed (all other fields are ignored)
- On TLB miss exceptions the TLB Page Descriptor register 0, `TLBPD0`, is updated with the VPN and ASID associated with the address that was the cause of the TLB miss exception. To aid the TLB miss handler, the global is cleared and the valid bit is set on TLB miss exceptions.

The layout of the register fields corresponds exactly to the ARC 700 page descriptors, see Page Descriptor Format. The operation of the TLB maintenance registers is not affected by the setting of the TLB-enable bit in the process identity register (PID [T]). All reserved bits in the page descriptor are set to zero.

TLB Index Register, TLBIndex

This register is set by the programmer to communicate the index for **TLBWrite** and **TLBRead** commands, and set by the hardware to communicate a result from the **TLBGetIndex** and **TLBProbe** commands. Bit 31 is set to indicate an error, i.e. a value of 0x8000.0000 or above indicates an error. See command descriptions for more information on usage. Writes to this register can be non-serializing. The address in `TLBIndex` register is mapped as shown below.



The Reserved field is set to zero. The following fields are described in more detail:

- Index, Read/Write
- E, Error Code, Read only

Index, Read/Write

This part of the register is set by the programmer to communicate the Index for **TLBWrite** and **TLBRead** commands, and set by the hardware to communicate a result from the **TLBGetIndex** and **TLBProbe** commands. If an error has occurred (E is set) then the Index contains the error code. See command description in the TLBCommand section for more information on usage.

Table 3 TLBIndex Addresses and Error Codes

Access Type	Address/Error Code	Description
JTLB	0x0-0xFF	This allows both **TLBWrite**, **TLBRead** to be performed on the JTLB RAM.
µITLB	0x200-0x203	This allows the entries in the µITLB to be read (**TLBRead**).
µDTLB	0x400-0x407	This allows the entries in the µDTLB to be read (**TLBRead**).
Error Code (E flag is set)	0x0	Failed operation.
Error Code (E flag is set)	0x1	Duplicate TLB entries

E, Error Code, Read only

This bit is set by the hardware when an error has occurred. Writes to this flag are ignored.

TLB Command Register, TLBCommand

This fully serializing register is used to initiate all transactions with the TLB. Data is communicated through the `TLBPD0`, `TLBPD1` and `TLBIndex` registers. TLB command operations can still be performed when MMU is disabled (when the T bit is 0 in the PID register).

The following commands are supported:

Table 4 TLB Command Register Command List

Cmd	Name	Description
0x1	**TLBWrite**	Write a TLB entry to the index location specified in `TLBIndex`. Also used to remove entries.
0x2	**TLBRead**	Read a TLB entry into `TLBPD0` or `TLBPD1` from the location specified in `TLBIndex`.
0x3	**TLBGetIndex**	Set `TLBIndex` to contain a suitable index location for the page descriptor in `TLBPD0` or `TLBPD1` or an error code
0x4	**TLBProbe**	Determine if a TLB entry is present that matches the virtual address supplied in `TLBPD0` or `TLBPD1`, and return its index location or an error code in `TLBIndex`.

TLBWrite Command

This command is used to load an entry into the TLB at the specified index location.

The operating system may determine an appropriate location for the entry by itself, or may ask the MMU hardware for a suggestion by using the **TLBGetIndex** command.

The **TLBWrite** command is also used to remove (shoot down) existing entries, by loading an entry with the V bit set false. The `TLBPD0` and `TLBPD1` register bits would typically be set to all zeros before issuing a **TLBWrite** command. The operating system may determine on its own the index of the entry to be removed, or may use the **TLBProbe** command to return an index that corresponds to a virtual address/ASID combination.

The **TLBWrite** command operation can still be performed when MMU is disabled (when the T bit is 0 in the PID register).

TLBWrite Usage

- Page descriptor to be loaded into the TLB is brought into the `TLBPD0` and `TLBPD1` auxiliary registers.
- `TLBIndex` contains the index location to which the entry is to be loaded.
- `TLBPD0` and `TLBPD1` auxiliary registers are unchanged after the **TLBWrite** operation
- If an invalid index value is supplied (out of range), the TLB Load request is ignored, and `TLBIndex` will be loaded with error flag E set and the Index field containing error code 0x0 (full value returned is 0x8000.0000).
- Invalid entries may be loaded (V=0). Such entries will not be considered during lookup operations, however this feature allows an entry to be invalidated and also allows an entire save/restore of the TLB contents to be performed.

TLBRead Command

This command is used to read an entry from the TLB, at the specified index location. The operating system may either determine the location to be read, or may use the **TLBProbe** command to obtain the location of an entry from a virtual address.

The **TLBRead** command operation can still be performed when MMU is disabled (when the T bit is 0 in the PID register).

TLBRead Usage

- The `TLBIndex` register contains the location from which the entry is to be read.
- The read and write permission bits (in total 4 bits) are always set to zeros when reading entries in the uITLB. Read and write permissions only apply to the µDTLB.
- The execution permission bits (2 bits) are always set to zeros when reading entries in the uDTLB. Execution permission only apply to the µITLB.
- The reserved bits are always set to zeros when reading entries in the Joint TLB (writes to these bits are ignored).
- `TLBPD0` and `TLBPD1` registers contain the TLB entry from the specified location. Entries in the TLB that are marked as invalid are returned as they appear in the TLB.
  - — If an invalid index value is supplied (out of range), the `TLBIndex` will be loaded with error flag E set and the Index field containing error code 0x0 (full value returned is 0x8000.0000), and the TLB Read operation returns an entry with all bits set to zero.

TLBGetIndex Command

This command is used to allow the hardware to provide a TLB index to which a new entry may be loaded. This has a number of benefits:

- The mechanism enables the creation of simple and fast TLB miss handlers that are independent of the size and associativity of the underlying TLB, and can rely on the hardware to manage the replacement algorithm.
- An operating system that is aware of the configuration of the TLB can implement a different or more sophisticated replacement algorithm than is supported by the hardware - at the cost of increasing the number of cycles taken during TLB misses.
- The handling of complex error conditions may be deferred to the operating system.

The **TLBGetIndex** command operation can still be performed when MMU is disabled (when the T bit is 0 in the PID register).

TLBGetIndex Usage

- The page descriptor to be loaded into the TLB is brought into the `TLBPD0` and `TLBPD1` special purpose registers.
  - — Certain implementations (e.g. fully associative) may not require the `TLBPD0` and `TLBPD1` registers to contain the new page descriptor that is to be loaded. However, in order to ensure that a TLB miss handler may be used with any TLB, the page descriptor should always be loaded before executing the TLBGetIndex operation.
- The `TLBIndex` register is loaded with the location to which the supplied page descriptor can be loaded.

— The replacement algorithm is pseudo-random.

— An index value is always returned, no error conditions are returned

- Invalid ways are selected first, before considering a pseudo-random generated victim.

TLBProbe Command

This command is used to check the TLB for an entry that matches a supplied virtual address, and return an index location or an error code.

The **TLBProbe** command operation can still be performed when MMU is disabled (when the T bit is 0 in the PID register).

TLBProbe Usage

- The V[17:0] field of the `TLBPD0` and `TLBPD1` register pair contains the virtual address for which the TLB is to be searched. The A[7:0] field of the `TLBPD0` and `TLBPD1` register pair contains the address space identifier (ASID) to be used for the search. All other bits in the `TLBPD0` and `TLBPD1` register pair are ignored.

— As a result of the command, the `TLBIndex` register is loaded with the index location at which the matching entry is located.

- If no matching entry is found in the TLB, the `TLBIndex` will be loaded with error flag E set and the Index field containing error code 0x0, (full value returned is 0x8000.0000).
- If more than one matching entry is found in the TLB, the `TLBIndex` register will be loaded with error flag E set and the Index field containing error code 0x1. The full value returned is 0x8000.0001.
- A *matching entry* is defined as a TLB entry for which:

— The valid (V) bit is set true, and

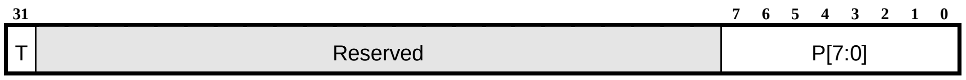
— The virtual address field V[17:0] matches exactly, and

— Either the ASID field A[7:0] matches exactly, or the global (G) bit is set

— No other information is used for matching - User/Kernel mode permissions and flag bits are not considered. The V bit in TLBPD0 is also ignored.

Process Identity Register, PID

The Process Identity register (`PID`) contains privilege bits that control permissions that can be optionally extended to a user mode task, an address space identifier (ASID) field used by the memory management system and compatibility mode bits. This is a fully serializing register.



The Reserved field is set to zero. The following fields are described in more detail:

- T, Global TLB Enable
- P[7:0], Address Space Identifier ASID

T, Global TLB Enable

The Global TLB Enable bit is used to enable or disable the MMU. When set to 0 the MMU is disabled, which means that all logical addresses are mapped directly to physical addresses. The MMU needs to be enabled (Global TLB Enable bit set to 1) in order for memory protection and cacheability to work on individual pages. Note that the Data Uncached region is always active even when the MMU is disabled. This field is set to 0x0 on reset.

P[7:0], Address Space Identifier ASID

The 8-bit Address Space Identifier (ASID) is set by the Operating System as the ASID of the currently executing process. The ASID is used by the Operating System and memory management hardware to allow physical pages to be mapped into many separate virtual address spaces. This field is set to 0x0 on reset.

Typically each independent task would have its own ASID value. This scheme is used to avoid the need to reload address mappings when context switching between tasks. The ASID in this register is checked against the ASID portion of a Page Descriptor (PD) unless the global bit, T, is set. Since there may be more than 256 tasks running at any one time, the Operating System manages the allocation and use of ASIDs.

NOTE The ASID is checked in both user and kernel mode - allowing the OS to run tasks in either mode.

Writes to the `PID` register should be made either from code running in un-translated memory or from code running from a page with the Global bit set (ASID is ignored). This ensures that the code page being accessed continues to be visible after the ASID is changed.

The processor ensures that the ASID update takes effect immediately after the SR instruction making the change.

NOTE A machine check exception causes the Global TLB enable to be cleared (set to zero).

Scratch Data Register, SCRATCH_DATA0

The `SCRATCH_DATA0` auxiliary register is a generic 32-bit scratch register that can be used in kernel mode only to store any data. The OS can for example use this register to hold the base address of the first level page descriptor table in order to speed up page table access. The default on reset is 0x0 and writes to the `SCRATCH_DATA0` are non-serializing.

Chapter 4 — Memory Management Exceptions

In this section:

Exceptions to Support Memory Management Functions

A number of exceptions are provided to support memory management functions:

- Instruction or Data TLB Miss
  - TLB lookup cannot locate an entry for the supplied virtual address
- TLB error
  - >1 matching entry during TLB lookup
- Protection violation
  - The access being attempted was not enabled by the protection flags in the TLB entry
- Unaligned access

An access was performed that violated the alignment constraints of the machine - accesses must be aligned to the size of the transaction.

For more information on the MMU related exceptions refer to the *ARCompact™ Programmer's Reference*.

The flow diagram (Figure 4) shows how exception conditions are detected.

Flowchart for TLB Lookups

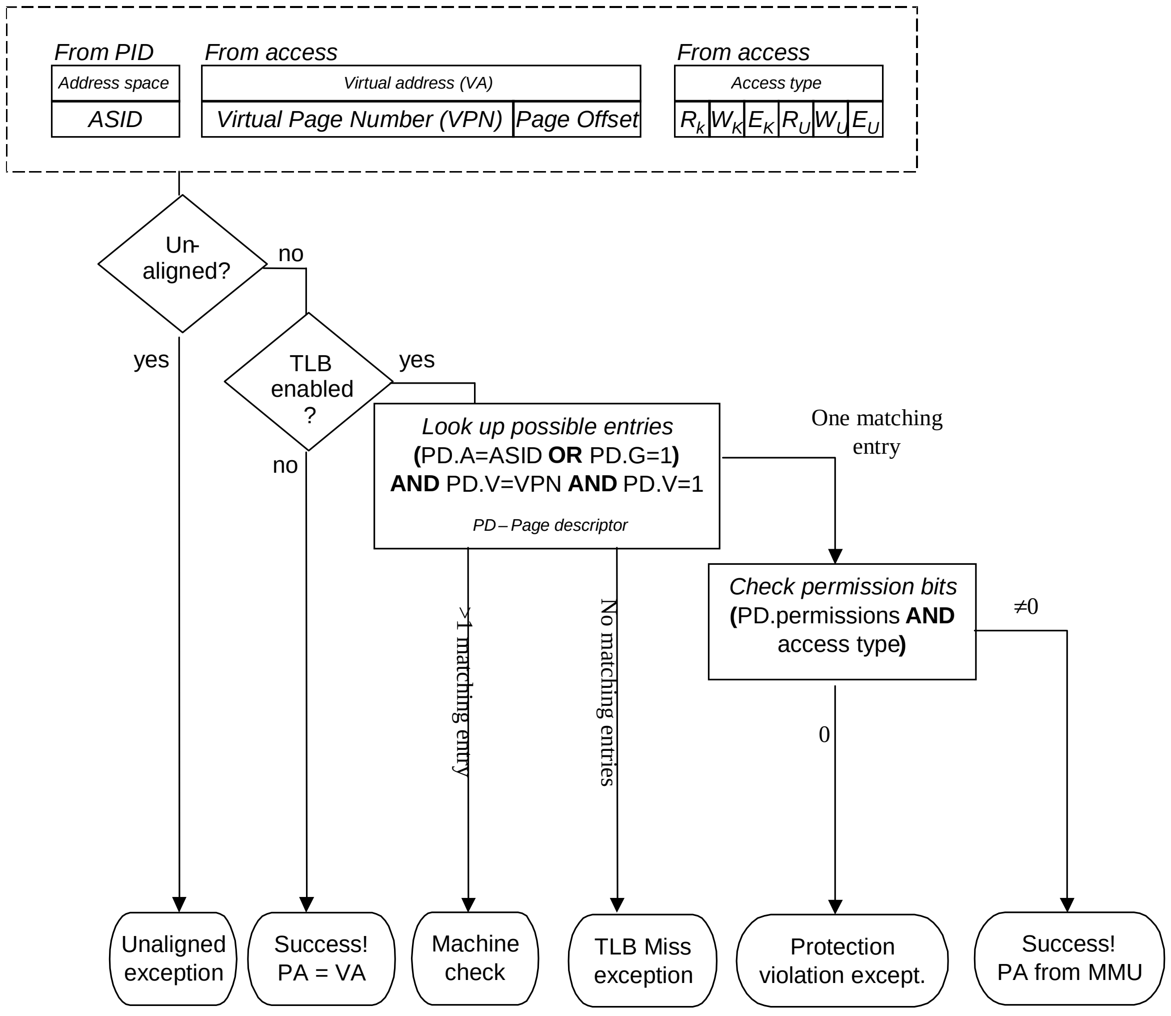


Figure 4 TLB Lookup Flowchart

Chapter 5 — Physical Address Calculation

In this section:

Calculation Process

When the Memory Management Unit (MMU) is enabled, physical addresses are calculated using the following inputs:

- Virtual address from the access (32 bits)
- Address space identifier (ASID) from the PID register
- TLB contents

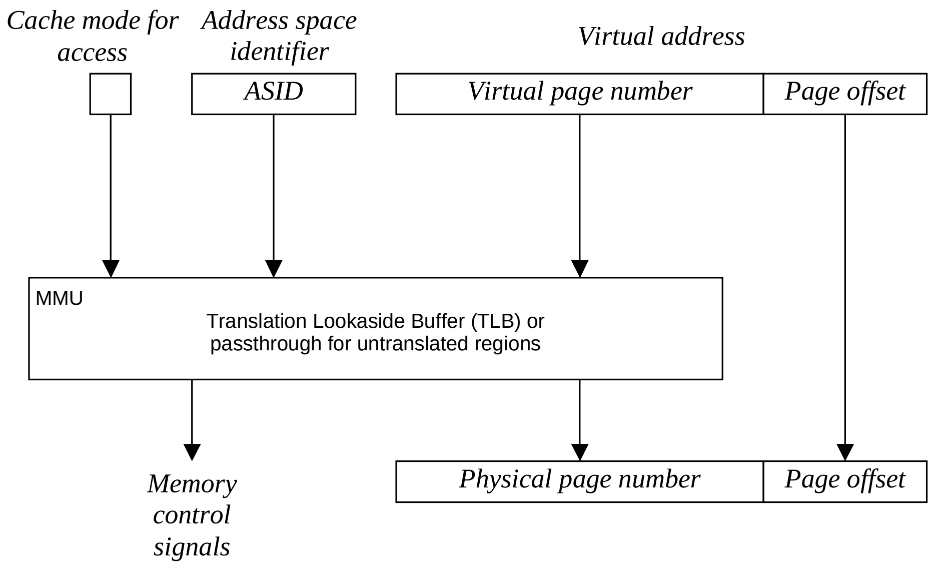


Figure 5 Physical Address Calculation

The outputs are as follows:

- The lower 13 bits (the page offset) come directly from the lower 13 bits of the virtual address supplied.
- The remaining bits (19) come directly from the Physical Page Number field $P_{18:0}$ in the matching TLB entry

The memory control signals are as follows:

- Cached/Uncached access
  - Determined from TLB entry and cache mode from original access.
  - Cached access permitted if the access requested a cached access and the TLB entry permits it. All other accesses are uncached - when either the instruction or the TLB entry specifies an uncached access.

Chapter 6 — Memory Configuration Examples

In this section:

Example Page Table Operations

Many modern operating systems implement demand-paged virtual memory systems. This method of managing memory enables a straightforward programming interface for application developers, and allows the operating system (OS) to dynamically manage the physical memory resources of the machine, and implement controls and protections for memory used by and shared between individual processes.

Properties of demand paged virtual memory systems include:

- Sharing
  - The physical memory attached to the machine can be shared between multiple processes simultaneously
  - More memory can be allocated than actually exists as physical memory in the machine, if disk storage is available
  - Areas of memory can be shared between two or more processes to allow for inter-process communications and data transfer, with process-specific protections
- Protection
  - Each process appears to have its own private address space
  - For any given process: Memory owned by the process is protected from accesses by other processes, and memory owned by other processes is protected from access by this process
- Translation
  - Address Translation maps program (*virtual*) address to hardware (*physical*) addresses
  - Infrequently used areas of memory can be swapped to disk until required

To implement common demand-paged virtual memory systems, certain hardware resources are required from the host processor - separate execution modes for user processes and the OS kernel, and a memory management unit providing address translation and memory protection.

All memory in the system is split into a number of regions, known as pages. Depending on the system, these pages can be of fixed or variable size. In the ARC 700 processor pages are 8 KB.

The operating system keeps track of the memory used by each process using a set of page tables. Each page in the address map of each process requires a *Page Table Entry* (PTE). Each process has its own address space - either the OS will support this through a single page table containing mappings for all address spaces, or by using a separate page table for each process.

The following sections provide further examples on page table operations:

- Memory Management Unit (MMU)
- Page Table Operations

Memory Management Unit (MMU)

The Memory Management Unit (MMU) provides hardware support and acceleration for address translation and protection. In effect the MMU acts as a cache into the page table - using a mechanism known as the *Translation Lookaside Buffer* (TLB). Like an instruction or data cache, the TLB is maintained to keep a subset of frequently used page table entries within the MMU, in order to allow for address translation and protection checks to be performed without delays. When a memory location is accessed for which the page table entry is not held in the TLB, the page table must be searched and the appropriate entry loaded - or if no matching page is found, an error condition generated.

In some systems the mechanism used to update the set of page table entries held in the TLB is provided by the MMU hardware. Other systems, including the ARC 700 processor, use a ‘software-managed’ TLB, where an exception handler is used to update the TLB entries from the page table. This approach enables a simpler hardware design, and greater flexibility for TLB management by software.

Page Table Operations

These sections give an illustration of MMU functions to support basic page table operations in a typical operating system. It is not intended to be an exhaustive list of all possible operations. Code is provided for illustrative purposes only.

- Add page table entry
- Remove page table entry
- Change page table entry
- TLB miss handlers
- Privilege Violation handlers

Add page table entry

When a new page table entry is added, no MMU operations are required. When a memory access is attempted to the new page, an exception will result and the page will be located and loaded by the TLB miss handler.

Remove page table entry

When a page table entry is removed, it is necessary to ensure that the MMU does not still contain the page in question.

The following function searches the MMU for a given address and removes it when present:

```
// mmu_shootdown_page:
//
//    Remove page from MMU from address and ASID
//
// Address : virtual address
// ASID : address space identifier (0-255)
//
void mmu_shootdown_page(long address, long asid) {
long result;

    // Load TLBPD0 with address and ASID
    //
    _sr((address & 0x7fffe000) + (asid & 0xff),TLBPD0);

    // Check for address in MMU with TLBProbe command
    //
    _sr(TLBProbe,TLBCommand);

    // Get result of probe
    //
    result = _lr(TLBIndex);
    // If a matching entry exists (top bit clear), remove it
    //
    //  - an update to the TLB will cause the uTLBs to be cleared
    //    thus ensuring the entry is cleared from there also.
    //
    if (!(result && 0x80000000)) {

        // Location of entry to be removed is already in TLBIndex
        //
        _sr(0,TLBPD0);
        _sr(0,TLBPD1);
        _sr(TLBWrite,TLBCommand);
    }
}
```

Change page table entry

When a page table entry is changed, it is necessary to ensure that the MMU contains the updated information.

The following function searches the MMU for a given page table entry and updates it if present. The OS could alternatively choose to remove an entry from the TLB after a change, thus forcing a reload by the TLB miss handler on the next access to the page.

```
// mmu_update_page:
//
//    Find page and update it if present.
//
// vaddress : virtual address
// asid     : address space identifier (0-255)
// global   : Global flag (0/1)
// paddress : physical address
// flags    : user and kernel flags (7 bits)
//
void mmu_update_page(long vaddress, long asid,
                     long global, long paddress, long flags) {
long result;

    // Check to see if page is present in the MMU
    //
    // Load TLBPD0 with address and ASID
    //
    _sr((vaddress & 0x7fffe000) + (asid & 0xff),TLBPD0);

    // Check for address in MMU with TLBProbe command
    //
    _sr(TLBProbe,TLBCommand);

    // Get result of probe
    //
    result = _lr(TLBIndex);

    // If a matching entry exists (top bit clear), reload it
    //
    //  - an update to the TLB will cause the uTLBs to be cleared
    //    thus ensuring the entry is cleared from there also.
    //
    if (!(result && 0x80000000)) {
        // Location of entry to be reloaded is already in TLBIndex
        //
        // Create TLBPD0
        //
        _sr(   (vaddress & 0x7fffe000)
             + (asid & 0xff)
             + ((global & 1)<<8)
             + (1 << 10), TLBPD0);

        // Create TLBPD1
        //
        _sr(   (paddress & 0xffffe000)
             + ((flags & 0x7f)<<2), TLBPD1);

        // Load entry into TLB
        //
        _sr(TLBWrite,TLBCommand);

    }
}
```

TLB miss handlers

A *TLB miss handler* is a performance-critical part of a software-managed MMU system, and would typically be written in assembler for maximum speed. The exact logic for the code depends on how the page tables are constructed in the particular operating system, but it is possible to describe the sequence of events required. The ARC 700 processor provides two vectors for TLB miss exceptions to allow for separate handling of TLB misses from instruction fetches and those from data accesses. However, these two vectors can be directed to the same handler if required.

The sequence of events for a TLB miss handler is illustrated in these steps:

- Save temp variables
- Get Page Table base address - for speed, the OS may choose to store it in SCRATCH_DATA0
- Get fault address from EFA register
- Search page table for the faulting address, in the current address space context - logic of the search is implementation-specific, dependent on the page table arrangement
- Based on the page table search:
  - If the requested page is not mapped into the address space of the process (i.e. it is not found in the page table), go to the page fault handler to deal with the error
  - If the requested page is mapped into the address space of the process, but the page is not loaded into physical memory, go to the page fault handler
  - If a mapping for the requested page is present in the page table, and the page itself is present in physical memory, continue to load the TLB entry
- At this point, the OS may choose to update the page table in order to keep track of which pages have been accessed, or to maintain other statistics.
- The TLB entry is constructed from the following data, extracted from the Page Table Entry:
  - Virtual Page Number

- Physical Page Number
- Address Space Identifier (ASID)
- User mode permission bits
- Kernel mode permission bits
- Valid bit
- Global bit

- The two halves of TLB entry are written into TLBPD0 and TLBPD1
- Execute TLBGetIndex command to get an index location in which to place the TLB entry. The command places an index value into TLBIndex, based on the data in TLBPD0 and TLBPD1
- Execute TLBLoad command to load the TLB entry in TLBPD0 and TLBPD1 at the location now in TLBIndex.
- Restore temp variables
- Exit

Privilege Violation handlers

In addition to the TLB miss handler, an operating system using the MMU must also provide handlers for *privilege violation* exceptions. These exceptions will occur when a program accesses a translated memory location in a way that is not allowed by the permission flags of the page, for example:

- Write attempt into *read-only* memory
- Jump into memory without execute permission

In most cases, a privilege violation in a user process would result in the process being terminated. However, there are some cases where privilege violation exceptions are used to assist with virtual memory operations.

In a demand-paged virtual memory system, pages are swapped between disk and physical memory. It is useful to determine whether a page in physical memory has become *dirty*, i.e. has been written since it was created or loaded from disk.

TLB entries in the ARC 700 MMU are never altered by the hardware once loaded - as a result, the MMU cannot set a flag to indicate that a write has taken place to a page.

In order to track dirty pages, a freshly created or loaded page is given read-only permissions in the TLB by the operating system. When the page is written to by the user program, an exception will be taken, at which point the OS can mark the page table entry as dirty. The TLB entry can be re-loaded with the proper read/write permissions and the program allowed to resume.

Example Arrangement

This is an example of the following arrangement:

- An operating system featuring a process model such as Linux
- The OS page tables, interrupt and exception handlers are located in un-translated memory above 0x80000000

- Three tasks - A, B and C
  - Two running in user mode (A and B)
  - One running in kernel mode (C)
- Tasks A and C share set of library functions
- Task C sends data to Task B via a shared memory block, to which only Task C has write access.
- Each task has its own stack and heap
- Each task is located in the same space in virtual memory (and hence the memory of other tasks is not visible)
- The operating system has exclusive access to memory mapped IO, and to its own stack and memory space - these are also located in un-translated memory about 0x80000000.

The diagrams on the following pages use the following shorthand for describing permissions (access mode flags are not shown):

- R,W,E: Kernel mode read/write/execute
- r,w,e: User mode read/write/execute
- g: Global access (ASID ignore)

Linux has the following rules for setting permissions for memory regions:

- Read access implies that execute access is granted
- Write access implies that read access is granted
  - Implying that execute access is also granted

This example assumes that the kernel mode permissions are set identically to the user mode permissions. If a debugging component of the operating system needs to write to code space, it is assumed that this component will need to set the appropriate write permission. An operating system designed for high reliability and availability would be likely to use the permission bits in a more sophisticated manner.

Operating System Private Space

In this example the OS has its own data stored in un-translated memory above 0x80000000, visible at all times when in kernel mode but invisible to user mode tasks:

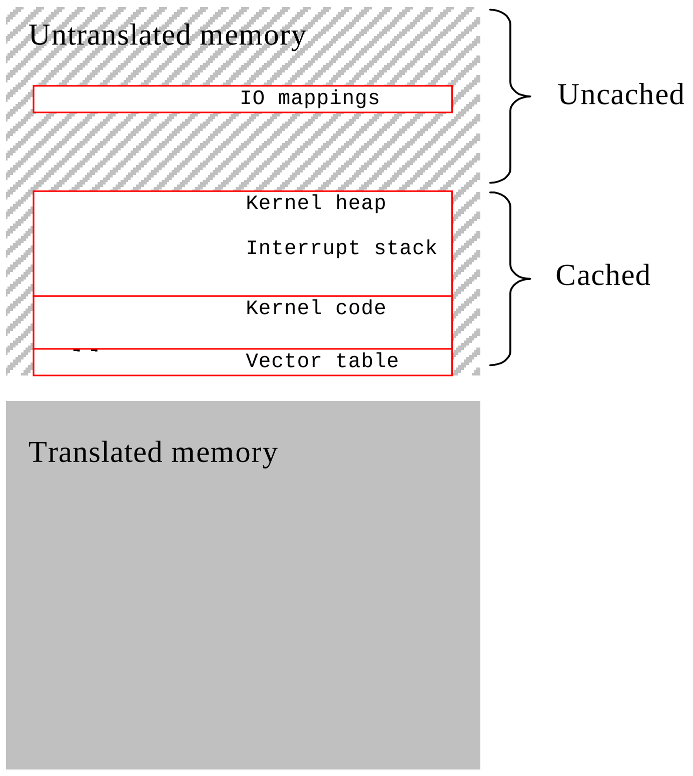


Figure 6 OS Private Space Memory Map

The pages are mapped into the address space at all times and the permissions prevent access from user mode tasks. Hence a user mode read, write or execute from these pages would be a protection violation and the appropriate exception generated. Clearly debugging systems or tasks would need to enable reads and writes to code space of user mode tasks in order to display disassemblies and to set and remove breakpoints.

When the processor is in kernel mode and a valid ASID is set, the address space will include not only the un-translated memory described above, but also the pages with matching ASID values. For example, if the kernel were entered whilst running task B, the memory space would be as follows:

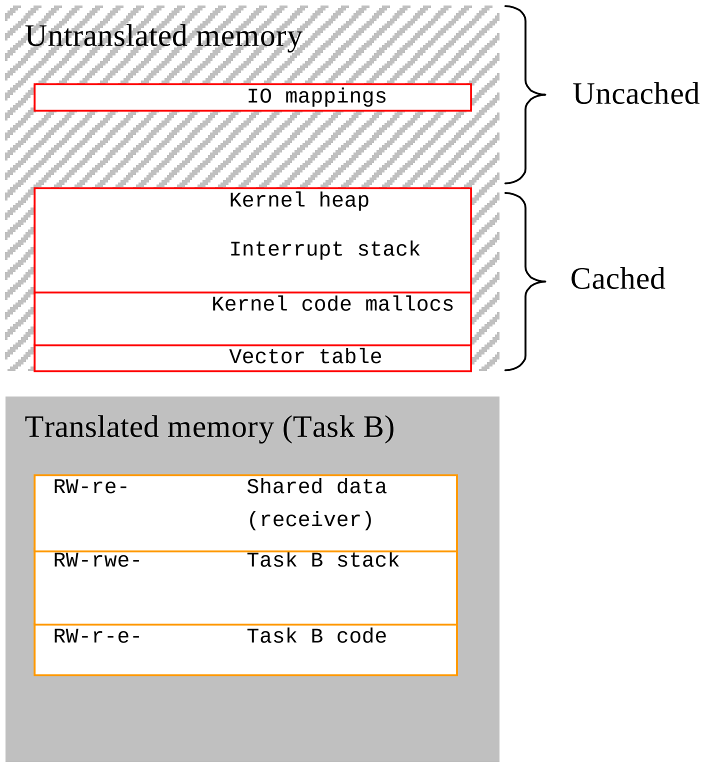


Figure 7 Task B Memory Map

In this example, kernel mode read/write access permissions are set on user task data areas in order to allow OS calls using kernel mode to take data from, and return data to the calling task's memory space.

User Mode Tasks

Two user-mode tasks are in the system - each has its own code and data area - mapped in the same location in the memory map in each case to prevent unwanted interaction between tasks.

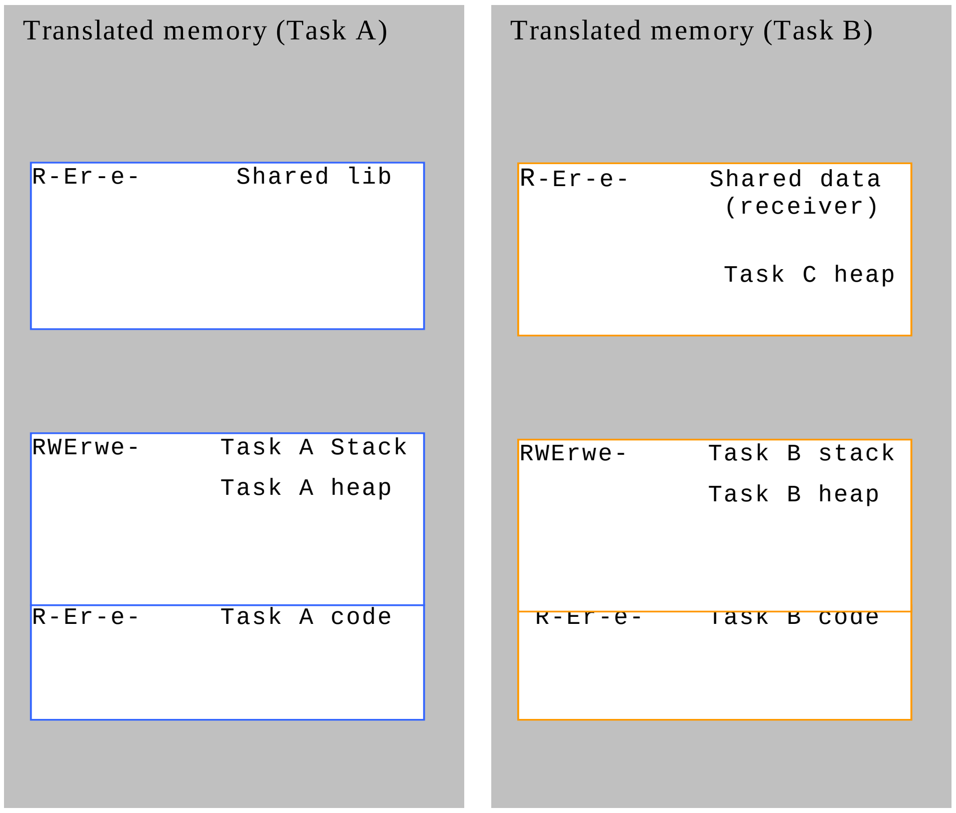


Figure 8 Task A and B Memory Maps

The un-translated memory region is not available to user mode tasks. Any access would cause a protection-violation exception, and hence this space not shown in the preceding memory map diagrams.

Kernel Mode Tasks

Some operating systems allow users to supply tasks (such as device drivers) that are to be run in kernel mode.

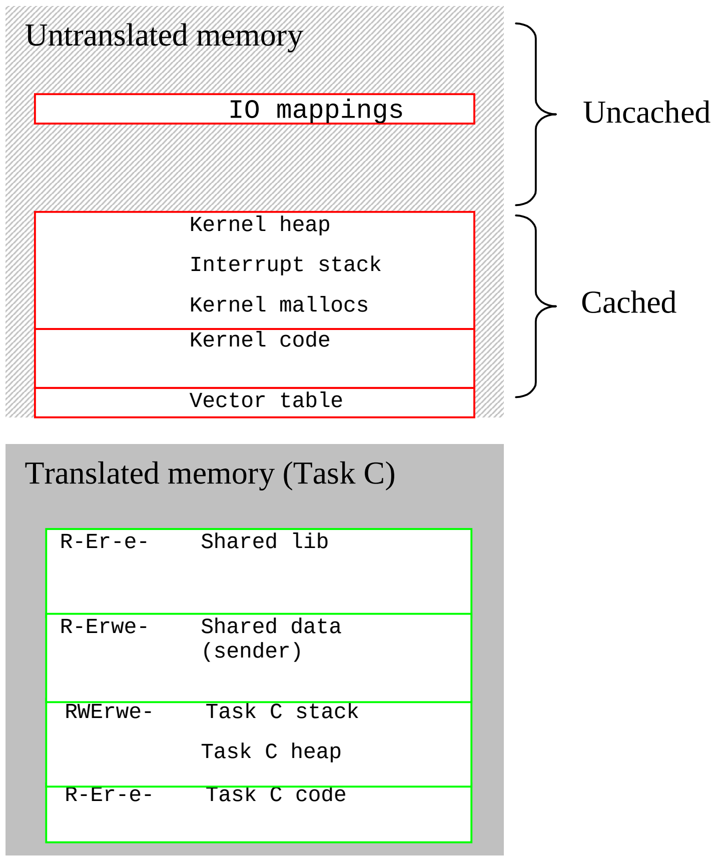


Figure 9 Task C Memory Map

In this example Task C is run in kernel mode. As such, it has access to its own memory spaces plus the un-translated memory space - which includes the memory mapped IO space.

Clearly the OS code and data areas in un-translated memory are not protected from erroneous writes from Task.

NOTE A malicious task running in kernel mode would have sufficient privileges to take over the entire system - hence the OS should only run trusted tasks or drivers in kernel mode

Shared Memory Regions

This example has two regions of memory shared between tasks - a shared data area and a shared library.

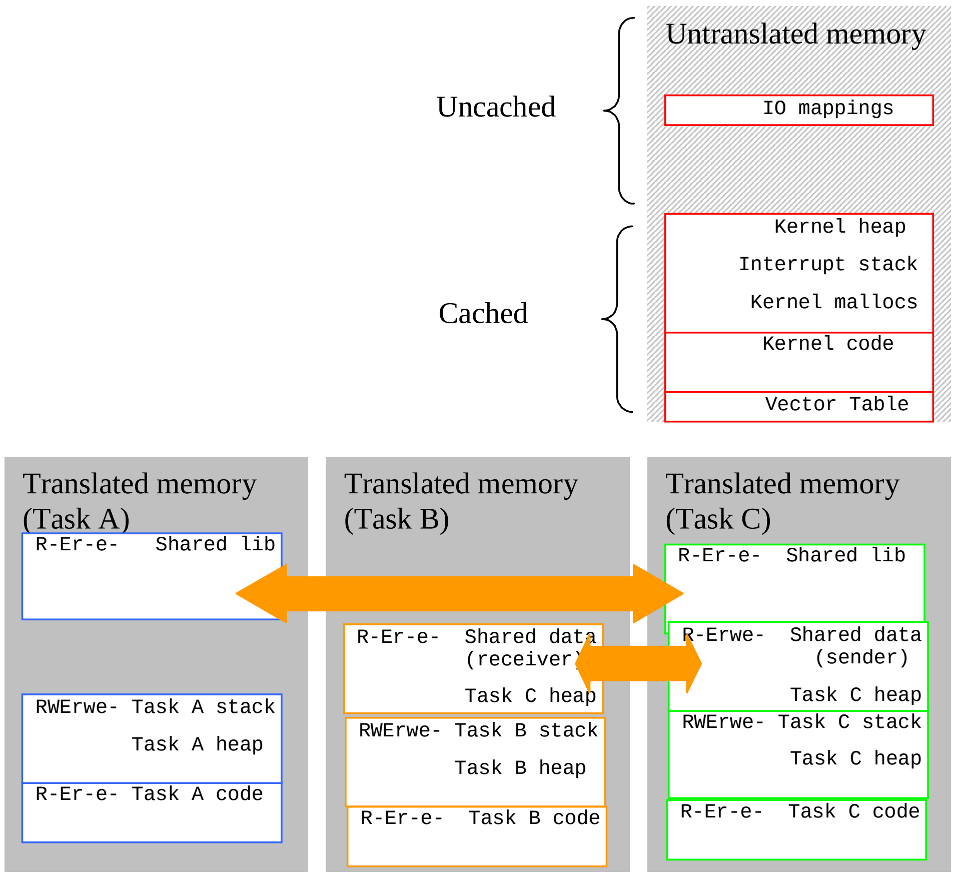


Figure 10 Shared Memory Regions

Since these shared areas of memory are shared between some tasks but not all tasks, they are not set to be globally accessible. Instead, multiple page table entries are created mapping to the same physical pages for the address spaces of each task requiring access.

The use of separate page table entries allows the access permissions for task to be set individually - allowing one task read-write access, and other tasks read-only access, for example.

In this case, task B only has read access to the shared data block, whereas task C (running in kernel mode) has both read and write access.

NOTE There is a restriction on how page mapping can be done for shared pages (see Restriction for Shared Pages).