Chapter 8
Caches

The word cache derives from the French verb *cacher*, “to hide”.The application of this word to a processor is obvious – a cache is where the processor stores instructions and data, hidden from the programmer and system. In many cases, it would be true to say that the cache is transparent to, or hidden from you. But very often, as we shall see, it is important to understand the operation of the cache in detail.

When the ARM architecture was first developed, the clock speed of the processor and the access speeds of memory were broadly similar. Processor cores today are much more complicated and can be clocked orders of magnitude faster. However, the frequency of the external buses and of memory devices has not scaled to the same extent. It is possible to implement small blocks of on-chip SRAM that can operate at the same speeds as the core, but such RAM is very expensive in comparison to standard DRAM blocks, that can have thousands of times more capacity. In many ARM processor-based systems, access to external memory will take tens or even hundreds of core cycles.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

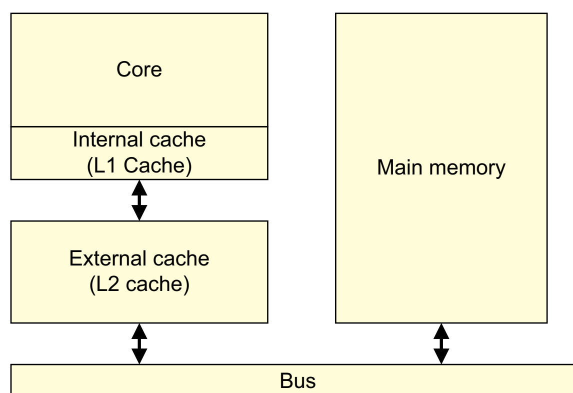

Figure 8-1 A basic cache arrangement

Essentially, a cache is a small, fast block of memory that (conceptually at least) sits between the core and main memory. It holds copies of items in main memory. Accesses to the cache memory happen significantly faster than those to main memory. Because the cache holds only a subset of the contents of main memory, it must store both the address of the item in main memory and the associated data. Whenever the core wants to read or write a particular address, it will first look for it in the cache. If it finds the address in the cache, it will use the data in the cache, rather than having to perform an access to main memory. This significantly increases the potential performance of the system, by reducing the effect of slow external memory access times. It also reduces the power consumption of the system, by avoiding the need to drive external signals.

Cache sizes are small relative to the overall memory used in the system. Larger caches make for more expensive chips. In addition, making an internal core cache larger can potentially limit the maximum speed of the core. Efficient use of this limited resource is a key part of writing efficient applications to run on a core.

On-chip SRAM can be used to implement caches, that hold temporary copies of instructions and data from main memory. Code and data have the properties of temporal and spatial locality. This means that programs tend to re-use the same addresses over time (temporal locality) and tend to use addresses that are near to each other (spatial locality). Code, for instance, can contain loops, meaning that the same code gets executed repeatedly or a function can be called multiple times. Data accesses (for example, to the stack) can be limited to small regions of memory. It is this fact that access to RAM by the core exhibits such locality, and is not truly random, that enables caches to be successful.

The write buffer is a block that decouples writes being done by the core when executing store instructions from the external memory bus. The core places the address, control and data values associated with the store into a set of hardware buffers. Like the cache, it sits between the core and main memory. This enables the core to move on and execute the next instructions without having to stop and wait for the slow main memory to actually complete the write operation.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.1 Why do caches help?

Caches speed things up, as we have seen, because program execution is not random. Programs tend to access the same sets of data repeatedly and execute the same sets of instructions repeatedly. By moving code or data into faster memory when it is first accessed, subsequent accesses to that code or data become much faster. The initial access that provided the data to the cache is no faster than normal. It is any subsequent accesses to the cached values that are faster, and it is from this that the performance increase derives. The core hardware will check all instruction fetches and data reads or writes in the cache, although obviously you must mark some parts of memory (those containing peripheral devices, for example) as non-cacheable. Because the cache holds only a subset of main memory, you require a way to determine (quickly) whether the address you are looking for is in the cache.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.2 Cache drawbacks

It might appear that caches and write buffers are automatically a benefit, as they speed up program execution. However, they also add some problems that are not present in an uncached core. One such drawback is that program execution time can become non-deterministic.

What this means is that, because the cache is small and holds only a subset of main memory, it fills rapidly as a program executes. When the cache is full, existing code or data must be removed to make room for new items. So, at any given time, it is not normally possible for an application to be certain whether or not a particular instruction or data item is to be found in the cache.

This means that the execution time of a particular piece of code can vary significantly. This can be something of a problem in hard real-time systems where strongly deterministic behavior is required.

Furthermore, you require a way to control how different parts of memory are accessed by the cache and write buffer. In some cases, you want the core to read up-to-date data from an external device, such as a peripheral. It would not be sensible to use a cached value of a timer peripheral, for example. Sometimes you want the core to stop and wait for a store to complete. So caches and write buffers give you some extra work to do.

Occasionally the contents of cache and external memory might not be the same, this is because the processor can update the cache contents, which have not yet been written back to main memory. Alternatively, an agent might update main memory after a core has taken its own copy. This is a problem of *coherency*. This can be a particular problem when you have multiple cores or memory agents like an external DMA controller. Coherency issues are described later in the book.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.3 Memory hierarchy

In computer science, a memory hierarchy refers to a hierarchy of memory types, with faster and smaller memories closer to the core and slower and larger memory farther away. In most systems, you can have secondary storage, such as disk drives and primary storage such as Flash, SRAM and DRAM. In embedded systems, you typically sub-divide this into on-chip and off-chip memory. Memory that is on the same chip (or at least in the same package) as the core will typically be much faster.

A cache can be included at any level in the hierarchy and can improve system performance where there is an access time difference between different parts of the memory system.

In ARM processor-based systems, level 1 (L1) caches are typically connected directly to the core logic that fetches instructions and handles load and store instructions. These are Harvard caches, that is, there are separate caches for instructions and for data that effectively appear as part of the core.

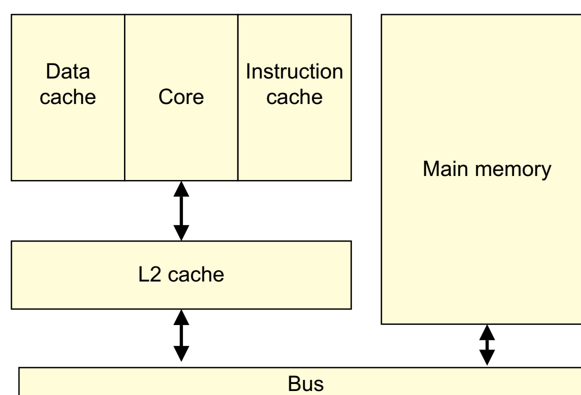

Figure 8-2 Typical Harvard cache

Over the years, the size of L1 caches has increased, because of SRAM size and speed improvements. At the time of writing, 16KB or 32KB cache sizes are most common, as these are the largest RAM sizes capable of providing single cycle access at a core speed of 1GHz or more.

Many ARM systems have, in addition, a level 2 (L2) cache. This is larger than the L1 cache (typically 256KB, 512KB or 1MB), but slower and unified (holding both instructions and data). It can be inside the core itself, or be implemented as an external block, placed between the core and the rest of the memory system. The ARM L2C-310 is an example of such an external L2 cache controller block.

In addition, cores can be implemented in clusters in which each core has its own level 1 cache. Such systems require mechanisms to maintain coherency between caches, so that when one core changes a memory location, that change is made visible to other cores that share that memory. This is described in more detail when we look at multi-core processors.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.4 Cache architecture

In a von Neumann architecture, a single cache is used for instruction and data (a unified cache). A modified Harvard architecture has separate instruction and data buses and therefore there are two caches, an instruction cache (I-cache) and a data cache (D-cache). In many ARM systems, you can have distinct instruction and data level 1 caches backed by a unified level 2 cache.

The cache requires to hold an address, some data and some status information. The top bits of the 32-bit address tells the cache where the information came from in main memory and is known as the *tag*. The total cache size is a measure of the amount of data it can hold; the RAMs used to hold tag values are not included in the calculation. The tag does, however, take up physical space in the cache.

It would be inefficient to hold one word of data for each tag address, so several locations are typically grouped together under the same tag. This logical block is commonly known as a cache *line*. The middle bits of the address, or *index*, identify the line. The index is used as address for the cache RAMs and does not require storage as a part of the tag. This will be covered in more detail later in this chapter. A cache line is said to be valid when it contains cached data or instructions, and invalid when it does not.

This means that the bottom few bits of the address (the offset) are not required to be stored in the tag – you require the address of a whole line, not of each byte within the line, so the five or six least significant bits will always be 0.

Associated with each line of data are one or more status bits. Typically, you will have a valid bit, that marks the line as containing data that can be used. (This means that the address tag represents some real value.) In a data cache you might also have one or more dirty bits that mark whether the cache line (or part of it) holds data that is not the same as (newer than) the contents of main memory.

8.4.1 Cache terminology

A brief summary of some of the terms used might be helpful:

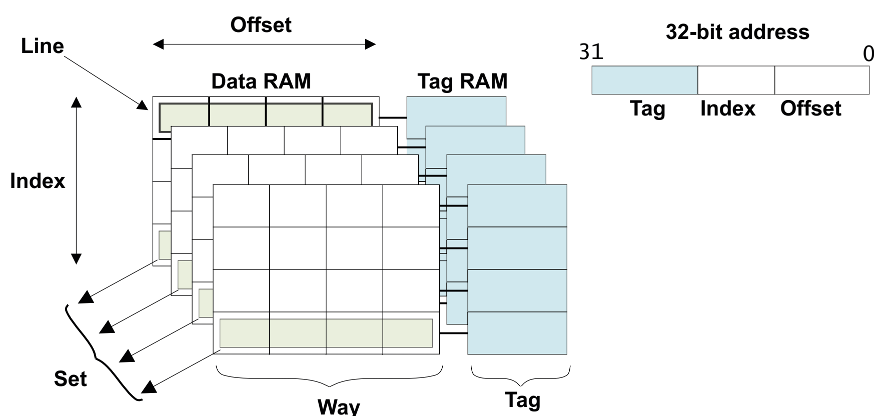

Figure 8-3 Cache terminology

- A *line* refers to the smallest loadable unit of a cache, a block of contiguous words from main memory.
- The *index* is the part of a memory address that determines in which line(s) of the cache the address can be found.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

- A *way* is a subdivision of a cache, each way being of equal size and indexed in the same fashion. The line associated with a particular index value from each cache way grouped together forms a *set*.
- The *tag* is the part of a memory address stored within the cache that identifies the main memory address associated with a line of data.

8.4.2 Direct mapped caches

There are different ways of implementing caches, the simplest of which is a *direct mapped* cache.

In a direct mapped cache, each location in main memory maps to a single location in the cache. However, as main memory is many times larger than the cache, many addresses will map to the same cache location. Figure 8-4 shows a small cache, with four words per line and four lines.

Figure 8-4 Direct mapped cache operation

This means that the cache controller will use two bits of the address (bits [3:2]) as the offset to select a word within the line and two bits of the address (bits [5:4]) as the index to select one of the four available lines. The remaining bits of the address (bits [31:6]) will be stored as a tag value.

Figure 8-5 Cache address

To look up a particular address in the cache, the hardware extracts the index bits from the address and reads the tag value associated with that line in the cache. If the two are the same and the valid bit indicates that the line contains valid data, it has a hit. It can then extract the data value from the relevant word of the cache line, using the offset and byte portion of the address. If the line contains valid data, but does not generate a hit (that is, the tag shows that the cache holds a different address in main memory) then the cache line is removed and is replaced by data from the requested address.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

It should be clear that all main memory addresses with the same value of bits [5:4] will map to the same line in the cache. Only one of those lines can be in the cache at any given time. This means a problem called *thrashing* can easily occur. Consider a loop that repeatedly accesses address `0x00, 0x40` and `0x80`, as in the following code:

```
void add_array(int *data1, int *data2, int *result, int size)
{
     int i;

     for (i=0 ; i<size ; i++) {
         result[i] = data1[i] + data2[i];
     }
}
```

In this code example, if `result`, `data1`, and `data2` are pointers to `0x00`, `0x40` and `0x80` respectively then this loop will cause repeated accesses to memory locations that all map to the same line in the basic cache, as shown in.Figure 8-4 on page 8-7.

- When you first read address `0x40`, it will not be in the cache and so a linefill takes place putting the data from `0x40` to `0x4F` into the cache.
- When you then read address `0x80`, it will not be in the cache and so a linefill takes place putting the data from `0x80` to `0x8F` into the cache – and in the process you lose the data from address `0x40` to `0x4F` from the cache.
- The result is written to `0x00`. Depending on the allocation policy this can cause another line fill. The data from `0x80` to `0x8F` might be lost.
- The same thing will happen on each iteration of the loop and our software will perform poorly. Direct mapped caches are therefore not typically used in the main caches of ARM cores, but you do see them in some places – for example in the branch target address cache of the ARM1136 processor.

Cores can have hardware optimizations for situations where the whole cache line is being written to. This is a condition that can take a significant proportion of total cycle time in some systems. For example, this can happen when `memcpy()`- or `memset()`-like functions that perform block copies or zero initialization of large blocks are executed. In such cases, there is no benefit in first reading the data values that will be over-written. This can lead to situations where the performance characteristics of the cache are different to what might normally be expected.

Cache allocate policies act as a hint to the core, they do not guarantee that a piece of memory will be read into the cache, and as a result, you should not rely on them.

8.4.3 Set associative caches

The main caches of ARM cores are always implemented using a set associative cache. This significantly reduces the likelihood of the cache thrashing seen with direct mapped caches, improving program execution speed and giving more deterministic execution. It comes at the cost of increased hardware complexity and a slight increase in power (because multiple tags are compared on each cycle).

With this kind of cache organization, the cache is divided into a number of equally-sized pieces, called *ways*. A memory location can then map to a way rather than a line. The index field of the address continues to be used to select a particular line, but now it points to an individual line in each way. Commonly, there are 2- or 4-ways, but some ARM implementations have used higher numbers.

Level 2 cache implementations (such as the ARM L2C-310) can have larger numbers of ways (higher associativity) because of their much larger size. The cache lines with the same index value are said to belong to a set. To check for a hit, you must look at each of the tags in the set.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

In Figure 8-6, a 2-way cache is shown. Data from address `0x00` (or `0x40`, or `0x80`) might be found in line 0 of either (but not both) of the two cache ways.

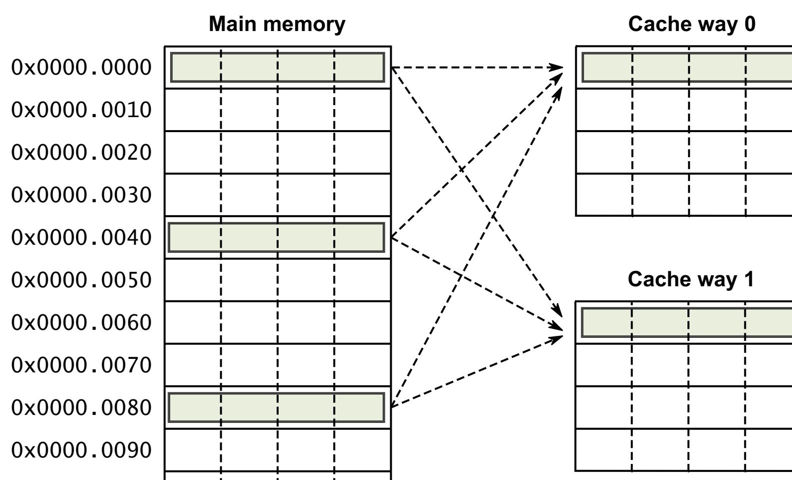

Figure 8-6 A 2-way set-associative cache

Increasing the associativity of the cache reduces the probability of thrashing. The ideal case is a fully associative cache, where any main memory location can map anywhere within the cache. However, building such a cache is impractical for anything other than very small caches (for example, those associated with MMU TLBs – see Chapter 9). In practice, performance improvements are minimal for Level 1 caches above 4-way associativity, with 8-way or 16-way associativity being more useful for larger level 2 caches.

8.4.4 A real-life example

Before going on to look at write buffers, let's consider an example that is more realistic than those shown in the previous two diagrams. Figure 8-7 on page 8-10 is a 4-way set associative 32KB data cache, with an 8-word cache line length. This kind of cache structure can be found on the Cortex-A7 or Cortex-A9 processors.

The cache line length is eight words (32 bytes) and you have 4-ways. 32KB divided by 4 (the number of ways), divided by 32 (the number of bytes in each line) gives you a figure of 256 lines in each way. This means that you require eight bits to index a line within a way (bits [12:5]). Here, you must use bits [4:2] of the address to select from the eight words within the line, though the number of bits which are required to index into the line depends on whether you are accessing a word, halfword, or byte. The remaining bits [31:13] in this case will be used as a tag.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

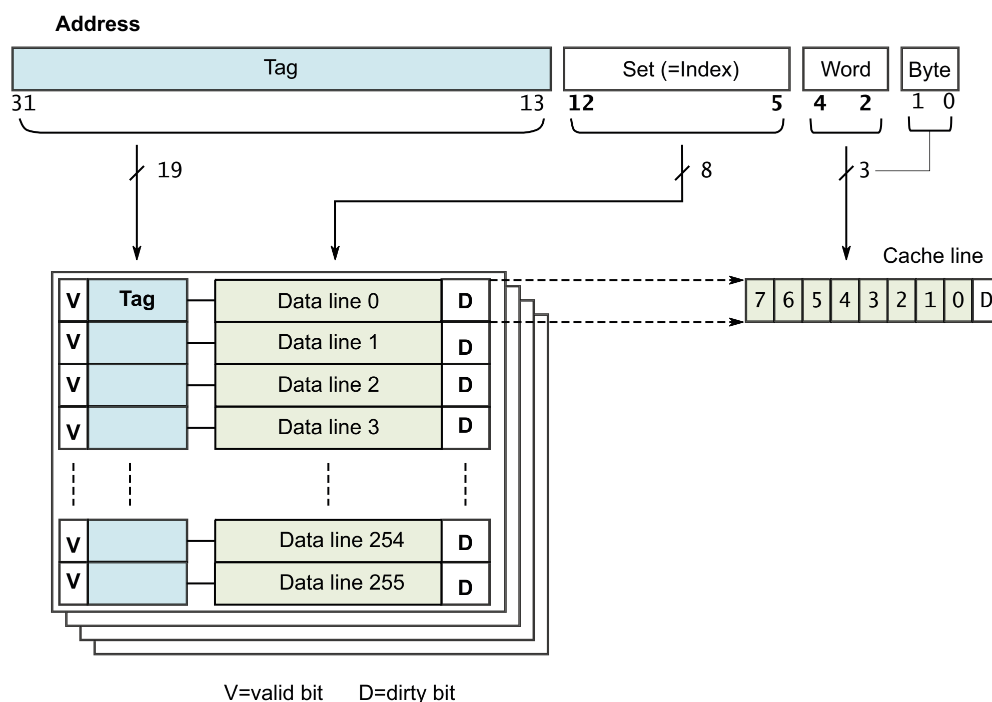

Figure 8-7 A 32KB 4-way set associative cache

8.4.5 Cache controller

This is a hardware block that has the task of managing the cache memory, in a way that is (largely) invisible to the program. It automatically writes code or data from main memory into the cache. It takes read and write memory requests from the core and performs the necessary actions to the cache memory or the external memory.

When it receives a request from the core it must check to see whether the requested address is to be found in the cache. This is known as a *cache look-up*. It does this by comparing a subset of the address bits of the request with tag values associated with lines in the cache. If there is a match (a hit) and the line is marked valid then the read or write will happen using the cache memory.

When the core requests instructions or data from a particular address, but there is no match with the cache tags, or the tag is not valid, a cache *miss* results and the request must be passed to the next level of the memory hierarchy – an L2 cache, or external memory. It can also cause a cache linefill. A cache linefill causes the contents of a piece of main memory to be copied into the cache. At the same time, the requested data or instructions are streamed to the core. This process happens transparently and is not directly visible to a software developer.

The core need not wait for the linefill to complete before using the data. The cache controller will typically access the *critical word* within the cache line first. For example, if you perform a load instruction that misses in the cache and triggers a cache linefill, the core first retrieves that part of the cache line which contains the requested data. This critical data is supplied to the core pipeline, while the cache hardware and external bus interface then read the rest of the cache line, in the background.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.4.6 Virtual and physical tags and indexes

This section assumes some knowledge of the address translation process. Readers unfamiliar with virtual addressing might want to revisit this section after reading Chapter 9.

A real-life example on page 8-9 was a little imprecise about specification of exactly which address is used to perform cache lookups. Early ARM processors such as the ARM720T or ARM926EJ-S processors used virtual addresses to provide both the index and tag values. This has the advantage that the core can do a cache look-up without the need for a virtual to physical address translation. The drawback is that changing the virtual to physical mappings in the system means that the cache must first be cleaned and invalidated, and this can have a significant performance impact. *Invalidating and cleaning cache memory* on page 8-17 goes into more detail about these terms.

ARM11 family processors use a different cache tag scheme. Here, the cache index is still derived from a virtual address, but the tag is taken from the physical address. The advantage of a physical tagging scheme is that changes in virtual to physical mappings do not now require the cache to be invalidated. This can have significant benefits for complex multi-tasking operating systems that can frequently modify translation table mappings. Using a virtual index has some hardware advantages. It means that the cache hardware can read the tag value from the appropriate line in each way in parallel without actually performing the virtual to physical address translation, giving a fast cache response. Such a cache is often described as *Virtually Indexed, Physically Tagged* (VIPT). Cache properties of Cortex-A series processors, including the use of these tagged caches are given in Table 8-1. Other properties of the Cortex-A series processors are listed in Table 2-3 on page 2-9.

Table 8-1 Cache features of Cortex-A series processors

	Processor					
	Cortex-A5	**Cortex-A7**	**Cortex-A8**	**Cortex-A9**	**Cortex-A12**	**Cortex-A15**
L2 Cache	External	Integrated	Integrated	External	Integrated	Integrated
L2 Cache size	-	128KB to 1MB[a]	0KB to 1MB[a]	-	256KB to 8MB	512KB to 4MB[a]
Cache Implementation (Data)	PIPT	PIPT	PIPT	PIPT	PIPT	PIPT
Cache Implementation (Instruction)	VIPT	VIPT	VIPT	VIPT	VIPT	PIPT
L1 Cache size (data)[a]	4K to 64K[a]	8KB to 64KB[a]	16/32KB[a]	16KB/32KB/64KB[a]	32KB	32KB
Cache size (Inst)[a]	4K to 64K[a]	8KB to 64KB[a]	16/32KB[a]	16KB/32KB/64KB[a]	32KB or 64KB	32KB
L1 Cache Structure	2-way set associative (Inst) 4-way set associative (Data)	2-way set associative (Inst) 4-way set associative (Data)	4-way set associative	4-way set associative (Inst) 4-way set associative (Data)	4-way set associative (Inst) 4-way set associative (Data)	2-way set associative (Inst) 2-way set associative (Data)
L2 Cache Structure	-	8-way set associative	8-way set associative	-	16-way set associative	16-way associative

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

Table 8-1 Cache features of Cortex-A series processors (continued)

	Processor					
	Cortex-A5	**Cortex-A7**	**Cortex-A8**	**Cortex-A9**	**Cortex-A12**	**Cortex-A15**
Cache line (words)	8	8	16	8	-	16
Cache line (bytes)	32	64	64	32	64	64
Error protection	None	None	L2 ECC	None	L1 None, L2 ECC	Optional for L1 and L2

a. Configurable

However, there is a drawback to a VIPT implementation. For a 4-way set associative 32KB or 64KB cache, bits [12] and [13] of the address are required to select the index. If 4KB pages are used in the MMU, bits [13:12] of the virtual address might not be equal to bits [13:12] of the physical address. There is therefore scope for potential cache coherency problems if multiple virtual address mappings point to the same physical address. This is resolved by placing certain restrictions on such multiple mappings that kernel translation table software must obey. This is described as a *page coloring* issue and exists on other processor architectures for the same reasons.

This problem is avoided by using a *Physically Indexed, Physically Tagged* (PIPT) cache implementation. The Cortex-A series of processors use such a scheme for their data caches. It means that page coloring issues are avoided, but at the cost of hardware complexity.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.5 Cache policies

There are a number of different choices that can be made in cache operation. Consider what causes a line from external memory to be placed into the cache (*allocation policy*) and how the controller decides which line within a set associative cache to use for the incoming data (*replacement policy*). What happens when the core performs a write that hits in the cache (*write policy*) must also be controlled.

8.5.1 Allocation policy

When the core performs a cache look-up and the address it wants is not in the cache, it must determine whether or not to perform a cache linefill and copy that address from memory.

- A *read allocate* policy allocates a cache line only on a read. If a write is performed by the core that misses in the cache, the cache is not affected and the write goes to the next level of the hierarchy.
- A *write allocate* policy allocates a cache line for *either* a read or write that misses in the cache (and so might more accurately be called a *read-write cache allocate* policy). For both memory reads that miss in the cache and memory writes that miss in the cache, a cache linefill is performed. This is typically used in combination with a *write-back* write policy on current ARM cores, as we shall see in *Write policy* on page 8-14.

8.5.2 Replacement policy

When there is a cache miss, the cache controller must select one of the cache lines in the set for the incoming data. The cache line selected is called the *victim*. If the victim contains valid, dirty data, the contents of that line must be written to main memory before new data can be written to the victim cache line. This is called *eviction*.

The *replacement policy* is what controls the victim selection process. The index bits of the address are used to select the set of cache lines, and the replacement policy selects the specific cache line from that set that is to be replaced.

- *Round-robin* or *cyclic* replacement means that you have a counter (the *victim counter*) that cycles through the available ways and cycles back to 0 when it reaches the maximum number of ways.
- *Pseudo-random* replacement randomly selects the next cache line in a set to replace. The victim counter is incremented in a pseudo-random fashion and can point to any line in the set.
- *Least Recently Used* (LRU) replacement is used to replace the cache line or page that was least recently used.

Most ARM processors support both Round-robin and Pseudo random policies. The Cortex-A15 processor also supports LRU.

A round-robin replacement policy is generally more predictable, but can suffer from poor performance in certain use cases and for this reason, the pseudo-random policy is often preferred.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.5.3 Write policy

When the core executes a store instruction, a cache lookup on the address(es) to be written is performed. For a cache hit on a write, there are two choices.

- *Write-through*. With this policy writes are performed to both the cache and main memory. This means that the cache and main memory are kept coherent. Because there are more writes to main memory, a write-through policy is slower than a write-back policy for some use cases, where the same area of memory is updated frequently. If large contiguous blocks of memory are being written to, and the writes can be buffered, it may be just as efficient to write-through. If the memory is not expected to be read from anytime soon (think large memory copies or memory initialization,) then it is better to not fill up the cache with such writes.
- *Write-back*. In this case, writes are performed only to the cache, and not to main memory. This means that cache lines and main memory can contain different data. The cache line holds newer data, and main memory contains older data (said to be *stale*). To mark these lines, each line of the cache has an associated *dirty* bit (or bits). When a write happens that updates the cache, but not main memory, the dirty bit is set. If the cache later evicts a cache line whose dirty bit is set (a *dirty line*), it writes the line out to main memory. Using a write-back cache policy can significantly reduce traffic to slow external memory and therefore improve performance and save power. However, if there are other agents in the system that can access memory at the same time as the core, you must consider coherency issues. These are described in *Cache coherency* on page 18-9.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.6 Write and Fetch buffers

A write buffer is a hardware block inside the core (but sometimes in other parts of the system as well), implemented using a number of buffers. It accepts address, data and control values associated with core writes to memory. When the core executes a store instruction, it might place the relevant details, such as the location to write to, the data to be written, and the transaction size into the buffer. The core does not have to wait for the write to be completed to main memory. It can proceed with executing the next instructions. The write buffer itself will drain the writes accepted from the core, to the memory system.

A write buffer can increase the performance of the system. It does this by freeing the core from having to wait for stores to complete. In effect, provided there is space in the write buffer, the write buffer is a way to hide latency. If the number of writes is low or well spaced, the write buffer will not become full. If the core generates writes faster than they can be drained to memory, the write buffer will eventually fill and there will be little performance benefit.

Some write buffers support write merging (also called write combining). They can take multiple writes (for example, a stream of writes to adjacent bytes) and merge them into one single burst. This can reduce the write traffic to external memory and therefore boost performance.

It will be obvious that sometimes the behavior of the write buffer is not what you want when accessing a peripheral, you might want the core to stop and wait for the write to complete before proceeding to the next step. Sometimes you really want a stream of bytes to be written and you don't want the stores to be combined.*ARM memory ordering model* on page 10-3, describes memory types supported by the ARM architecture and how to use these to control how caches and write buffers are used for particular devices or parts of the memory map.

Similar components, called fetch buffers, can be used for reads in some systems. In particular, cores typically contain prefetch buffers that read instructions from memory ahead of them actually being inserted into the pipeline. In general, such buffers are transparent to you. Some possible hazards associated with this will be considered when we look at memory ordering rules.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.7 Cache performance and hit rate

The *hit rate* is defined as the number of cache hits divided by the number of memory requests made to the cache during a specified time, normally calculated as a percentage. Similarly, the *miss rate* is the number of total cache misses divided by the total number of memory requests made to the cache. One might also calculate the number of hits or misses on reads or writes only.

Clearly, a higher hit rate will generally result in higher performance. It is not really possible to quote example figures for typical software, the hit rate is very dependent on the size and spatial locality of the critical parts of the code or data operated on and of course, the size of the cache.

There are some simple rules that can be followed to give better performance. The most obvious of these is to enable caches and write buffers and to use them wherever possible (typically for all parts of the memory system that contain code and more generally for RAM and ROM, but not peripherals). Performance will be considerably increased in Cortex-A series processors if instruction memory is cached. Placing frequently accessed data together in memory can also be helpful. For example, a frequently accessed array might benefit from having a base address at the start of a cache line.

Fetching a data value in memory involves fetching a whole cache line; if none of the other words in the cache line will be used, there will be little or no performance gain. This can be mitigated by accessing data in a manner that is ‘cache-friendly’.For instance, accesses to sequential addresses, for example, accessing a row of an array, benefit from cache behavior, Non-predictable or non-sequential access patterns, for example, linked lists, do not.

Smaller code might cache better than larger code and this can sometimes give seemingly paradoxical results. For example, a piece of C code might fit entirely within cache when compiled for Thumb (or for the smallest size) but not when compiled for ARM (or for maximum performance) and as a consequence can actually run faster than the more optimized version. Cache considerations are described in much more detail in Chapter 17 *Optimizing Code to Run on ARM Processors*.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.8 Invalidating and cleaning cache memory

Cleaning and invalidation can be required when the contents of external memory have been changed and you want to remove stale data from the cache. It can also be required after MMU related activity such as changing access permissions, cache policies, or virtual to physical address mappings.

The word *flush* is often used in descriptions of clean and invalidate operations. ARM generally uses only the terms *clean* and *invalidate*.

- Invalidation of a cache or cache line means to clear it of data. This is done by clearing the valid bit of one or more cache lines. The cache must always be invalidated after reset as its contents will be undefined. If the cache contains dirty data, it is generally incorrect to invalidate it. Any updated data in the cache from writes to write-back cacheable regions would be lost by simple invalidation.
- Cleaning a cache or cache line means writing the contents of dirty cache lines out to main memory and clearing the dirty bit(s) in the cache line. This makes the contents of the cache line and main memory coherent with each other. This is only applicable for data caches in which a write-back policy is used. Cache invalidate, and cache clean operations can be performed by cache set, or way, or by virtual address.

Copying code from one location to another (or other forms of self-modifying code) might require you either to clean and/or to invalidate the cache. The memory copy code will use load and store instructions and these will operate on the data side of the core. If the data cache is using a write-back policy for the area to which code is written, it is necessary to clean that data from the cache before the code can be executed. This ensures that the instructions stored as data go out into main memory and are then available for the instruction fetch logic. In addition, if the area to which code is written was previously used for some other program, the instruction cache could contain stale code (from before main memory was re-written). Therefore, it might also be necessary to invalidate the instruction cache before branching to the newly copied code.

The commands to either clean or invalidate the cache are CP15 operations. They are available only to privileged code and cannot be executed in User mode. In systems where the TrustZone Security Extensions are in use, there can be hardware limitations applied to non-secure use of some of these operations.

CP15 instructions exist that will clean, invalidate, or clean *and* invalidate level 1 data or instruction caches. Invalidation without cleaning is safe only when it is known that the cache cannot contain dirty data – for example a Harvard instruction cache, or when the data is going to be overwritten, and you don't care about losing the previous values. You can perform the operation on the entire cache, or on individual lines. These individual lines can be specified either by giving a virtual address to be cleaned or to be invalidated, or by specifying a line number in a particular set, in cases where the hardware structure is known. The same operations can be performed on the L2 or outer caches and we will look at this in *Level 2 cache controller* on page 8-22. A typical example of such code can be found in *Setting up caches, MMU and branch predictors* on page 13-3.

Example 8-1 Preparing the caches

```
setup_caches
     MRC p15, 0, r1, c1, c0, 0          ; Read System Control Register (SCTLR)
     BIC r1, r1, #1                     ; mmu off
     BIC r1, r1, #(1 << 12)             ; i-cache off
     BIC r1, r1, #(1 << 2)              ; d-cache & L2-$ off
     MCR p15, 0, r1, c1, c0, 0          ; Write System Control Register (SCTLR)
 ;-----------------------------------------------
```


Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

```
; 1.MMU, L1$ disable
;-----------------------------------------------
     MRC p15, 0, r1, c1, c0, 0          ; Read System Control Register (SCTLR)
     BIC r1, r1, #1                     ; mmu off
     BIC r1, r1, #(1 << 12)             ; i-cache off
     BIC r1, r1, #(1 << 2)              ; d-cache & L2-$ off
     MCR p15, 0, r1, c1, c0, 0          ; Write System Control Register (SCTLR)
;-----------------------------------------------
; 2. invalidate: L1$, TLB, branch predictor
;-----------------------------------------------
     MOV     r0, #0
     MCR     p15, 0, r0, c7, c5, 0      ; Invalidate Instruction Cache
     MCR     p15, 0, r0, c7, c5, 6      ; Invalidate branch prediction array
     MCR     p15, 0, r0, c8, c7, 0      ; Invalidate entire Unified Main TLB
     ISB                                ; instr sync barrier
;-----------------------------------------------
; 2.a. Enable I cache + branch prediction
;-----------------------------------------------
     MRC     p15, 0, r0, c1, c0, 0          ; System control register
     ORR     r0, r0, #1 << 12               ; Instruction cache enable
     ORR     r0, r0, #1 << 11               ; Program flow prediction
     MCR     p15, 0, r0, c1, c0, 0          ; System control register
;-----------------------------------------------
```

Of course, these operations will be accessed through kernel code – in GCC on Linux, you will use the `__clear_cache()` function implemented in `arch/arm/mm/cache-v7.S`.

```
void __clear_cache(char* beg, char* end);
```

The start address (char* beg) is inclusive, while the end address (char* end) is exclusive.

Equivalent functions exist in other operating systems, Google Android has `cacheflush()`, for example.

A common situation where cleaning or invalidation can be required is DMA (Direct Memory Access). When it is required to make changes made by the core visible to external memory, so that it can be read by a DMA controller, it might be necessary to clean the cache. When external memory is written by a DMA controller and it is necessary to make those changes visible to the core, the affected addresses must be invalidated in the cache.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.9 Point of coherency and unification

For set/way based clean and invalidate, the operation is performed on a specific level of cache. For operations that use a virtual address, the architecture defines two conceptual points:

Point of Coherency (PoC)

For a particular address, the PoC is the point at which all blocks, for example, cores, DSPs, or DMA engines, that can access memory are guaranteed to see the same copy of a memory location. Typically, this will be the main external system memory.

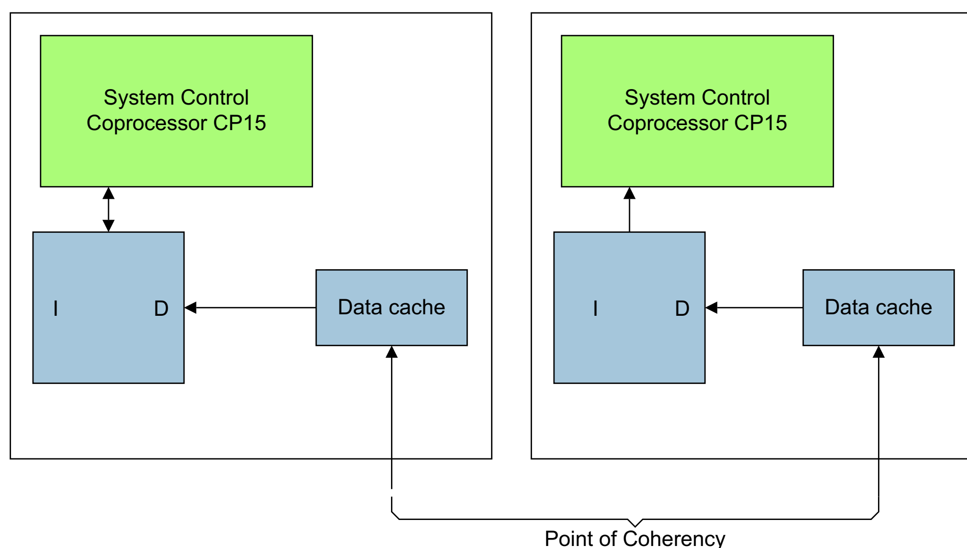

Figure 8-8 Point of Coherency

Point of Unification (PoU)

The PoU for a core is the point at which the instruction and data caches of the core are guaranteed to see the same copy of a memory location. For example, a unified level 2 cache would be the point of unification in a system with Harvard level 1 caches and a TLB for cacheing translation table entries. If no external cache is present, main memory would be the Point of unification.

In the Cortex-A9 processor the PoC and PoU is essentially the same place, at the L2 interface.

Since the Cortex-A8 processor incorporates a L2 cache under CP15 control PoU and PoC are in different places, PoU is in the L2 cache and PoC is outside the L2 interfaces.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

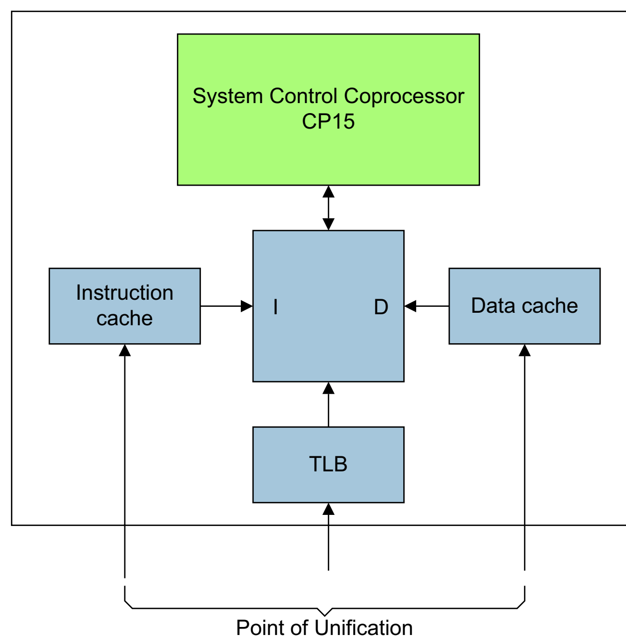

Figure 8-9 Point of Unification

Readers unfamiliar with the terms hardware translation table walk or Translation Lookaside Buffer (TLB) will find these described in Chapter 9. If no external cache is present, main memory would be the PoU.

In the case of a cluster, or a big.LITTLE combination, the PoU is where instruction and data caches and translation table walks of all the cores within the cluster are guaranteed to see the same copy of a memory location.

Knowledge of the PoU enables self-modifying code to ensure future instruction fetches are correctly made from the modified version of the code. They can do this by using a two-stage process:

- Clean the relevant data cache entries by address.
- Invalidate instruction cache entries by address.

In addition, the use of memory barriers will be required.

8.9.1 Example code for cache maintenance operations

The following code illustrates a generic mechanism for cleaning the entire data or unified cache to the point of coherency.

Note

In the case of a cluster where multiple cores share a cache before the point of coherency, running this sequence on multiple cores results in the operations being repeated on the shared cache

---

```
    MRC p15, 1, R0, c0, c0, 1   ; Read CLIDR into R0
    ANDS R3, R0, #0x07000000
    MOV R3, R3, LSR #23         ; Cache level value (naturally aligned)
    BEQ Finished
    MOV R10, #0
Loop1
    ADD R2, R10, R10, LSR #1    ; Work out 3 x cache level
    MOV R1, R0, LSR R2          ; bottom 3 bits are the Cache type for this level
```


Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

```
     AND R1, R1, #7              ; get those 3 bits alone
     CMP R1, #2
     BLT Skip                    ; no cache or only instruction cache at this level
     MCR p15, 2, R10, c0, c0, 0  ; write CSSELR from R10
     ISB                         ; ISB to sync the change to the CCSIDR
     MRC p15, 1, R1, c0, c0, 0   ; read current CCSIDR to R1
     AND R2, R1, #7              ; extract the line length field
     ADD R2, R2, #4              ; add 4 for the line length offset (log2 16 bytes)
     LDR R4, =0x3FF
     ANDS R4, R4, R1, LSR #3     ; R4 is the max number on the way size (right aligned)
     CLZ R5, R4                  ; R5 is the bit position of the way size increment
     MOV R9, R4                  ; R9 working copy of the max way size (right aligned)
Loop2
     LDR R7, =0x00007FFF
     ANDS R7, R7, R1, LSR #13    ; R7 is the max num of the index size (right aligned)
Loop3
     ORR R11, R10, R9, LSL R5    ; factor in the way number and cache number into R11
     ORR R11, R11, R7, LSL R2    ; factor in the index number
     MCR p15, 0, R11, c7, c10, 2 ; DCCSW, clean by set/way
     SUBS R7, R7, #1             ; decrement the index
     BGE Loop3
     SUBS R9, R9, #1             ; decrement the way number
     BGE Loop2

Skip
     ADD R10, R10, #2            ; increment the cache number
     CMP R3, R10
     BGT Loop1
     DSB
Finished
```

Similarly, you can use the clean data cache entry and invalidate TLB operations to ensure that all writes to the translation tables are visible to the MMU.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.10 Level 2 cache controller

At the start of this chapter, we briefly described the partitioning of the memory system and explained how many systems have a multi-level cache hierarchy. The Cortex-A5 and Cortex-A9 processors, however, do not have an integrated level 2 cache. Instead, the system designer can opt to connect another cache controller, such as the ARM L2 cache controller (L2C-310) outside of the processor instance.

The L2C-310 cache controller can support a cache of up 8MB in size, with a set associativity of between four and sixteen ways. The size and associativity are fixed by the SoC designer. The level 2 cache can be shared between multiple cores, or indeed between the core and other agents, such as a graphics processor. It is possible to lockdown cache data on a per-master per-way basis, enabling management of cache sharing between multiple components.

8.10.1 Level 2 cache maintenance

Virtual and physical tags and indexes on page 8-11 described how you might require the ability either to clean or invalidate some or all of a cache. This can be done by writing to memory-mapped registers within the L2 cache controller in the case where the cache is external to the core, or through CP15, where the level 2 cache is implemented inside the core. The registers themselves are not cached, which makes this feasible. Where such operations are performed by having the core perform memory-mapped writes, the core must have a way of determining when the operation is complete. It does this by polling an additional memory-mapped register within the L2 cache controller.

The ARM L2C-310 Level 2 cache controller operates only on physical addresses. Therefore, to perform cache maintenance operations, it might be necessary for the program to perform a virtual to physical address translation. The L2C-310 provides a *cache sync* operation that forces the system to wait for pending operations to complete.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

8.11 Parity and ECC in caches

So-called *soft errors* are an increasing concern. Smaller transistor geometries and lower voltages give circuits an increased sensitivity to perturbation by cosmic rays and other background radiation, alpha particles from silicon packages, or from electrical noise. This is particularly true for memory devices that rely on storing small amounts of charge and that also occupy large proportions of total silicon area. In some systems, mean-time-between-failure could be measured in seconds if appropriate protection against soft errors was not employed.

The ARM architecture provides support for parity and *Error Correcting Code* (ECC) in the caches. Parity means that there is an additional bit that marks whether the number of bits with the value one is even or odd, depending on the scheme chosen. This provides a simple check against single bit errors.

An ECC scheme enables detection of multiple bit failures and possible recovery from soft errors, but recovery calculations can take several cycles. Implementing a core that is tolerant of level 1 cache RAM accesses taking multiple clock cycles significantly complicates the design. ECC is therefore more commonly used only on blocks of memory (for example, the Level 2 cache), outside the core. The Cortex-A15, however, supports ECC and parity inside the core.

Parity is checked on reads and writes, and can be implemented on both tag and data RAMs. Parity mismatch generates a prefetch or data abort exception, and the fault status address registers are updated appropriately.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

Chapter 9
The Memory Management Unit

An important function of a *Memory Management Unit* (MMU) is to enable you to manage tasks as independent programs running in their own private virtual memory space. A key feature of such a virtual memory system is address relocation, or the translation of the virtual address issued by the processor to a physical address in main memory.

The ARM MMU is responsible for translating addresses of code and data from the virtual view of memory to the physical addresses in the real system. The translation is carried out by the MMU hardware and is transparent to the application. In addition, the MMU controls such things as memory access permissions, memory ordering and cache policies for each region of memory.

In multi-tasking embedded systems, we typically require a way to partition the memory map and assign permissions and memory attributes to these regions of memory. In situations where we are running more complex operating systems, like Linux, we require even greater control over the memory system.

The MMU enables tasks or applications to be written in a way that requires them to have no knowledge of the physical memory map of the system, or about other programs that might be running simultaneously. This enables you to use the same virtual memory address space for each program. It also lets you work with a contiguous virtual memory map, even if the physical memory is fragmented. This virtual address space is separate from the actual physical map of memory in the system. Applications are written, compiled and linked to run in the virtual memory space. Virtual addresses are those used by you, and the compiler and linker, when placing code in memory. Physical addresses are those used by the actual hardware system.

It is the responsibility of the operating system to program the MMU to translate between these two views of memory. Figure 9-1 on page 9-2 shows an example system, illustrating the virtual and physical views of memory. Different processors and/or devices in a single system might have different virtual and physical address maps, for example, some multi-core boards and PCI devices.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

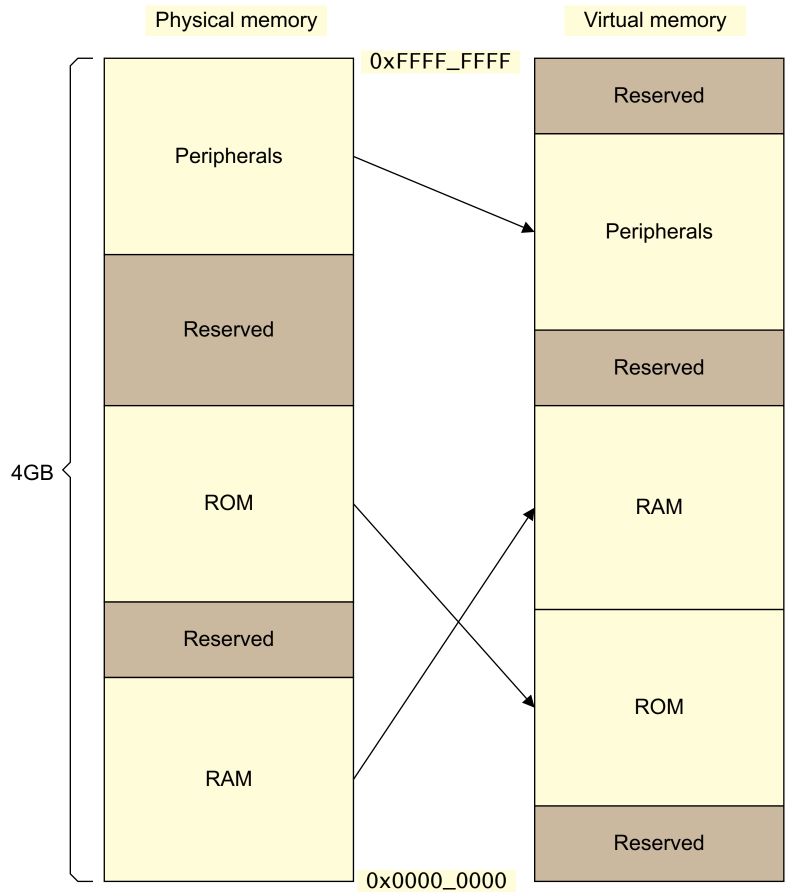

Figure 9-1 Virtual and physical memory

When the MMU is disabled, all virtual addresses map directly to the corresponding physical address (a *flat mapping*). If the MMU cannot translate an address, it generates an abort exception on the processor and provides information to the processor about what the problem was. This feature can be used to map memory or devices on-demand, one page at a time.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

9.1 Virtual memory

The MMU enables you to build systems with multiple virtual address maps. Each task can have its own virtual memory map. The OS kernel places code and data for each application in physical memory, but the application itself does not require the location.

The address translation carried out by the MMU is done using *translation tables*. These are tree-shaped table data structures created by software in memory, that the MMU hardware traverses to accomplish virtual address translation.

Note

In the ARM architecture, the concept referred to in generic computer terminology as *page tables* has a more specific meaning. The ARM architecture uses multi-level page tables, and defines *translation tables* as a generic term for all of these. An entry in a translation table contains all the information required to translate a page in virtual memory to a page in physical memory. The specific mechanism of traversal and the table format are configurable by software and are explained later.

Translation table entries are organized by virtual address. In addition to describing the translation of that virtual page to a physical page, they also provide access permissions and memory attributes necessary for that page.

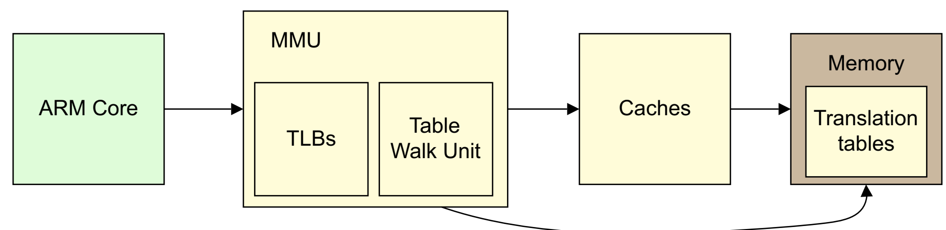

Figure 9-2 The Memory Management Unit

Addresses generated by the core are virtual addresses. When the MMU is enabled all memory accesses made by the core pass through it. The MMU essentially replaces the most significant bits of this virtual address with some other value, to generate the physical address (effectively defining a base address of a piece of memory).The same translation tables are used to define the translations and memory attributes that apply to both instruction fetches and to data accesses. Dedicated hardware within the MMU enables it to read the translation tables in memory. This process is known as *translation table walking*.

9.1.1 Configuring and enabling the MMU

Before the MMU is enabled, the translation tables must be written to memory. The TTBR register must be set to point to the tables. The following code sequence can then be used to enable the MMU:

```
MRC p15, 0, R1, c1, C0, 0       ;Read control register
ORR R1, #0x1                    ;Set M bit
MCR p15, 0,R1,C1, C0,0          ;Write control register and enable MMU
```

Care must be taken if enabling the MMU changes the address mapping of the region in which code is currently being executed. Barriers (See *Memory barriers* on page 10-6) may be necessary to ensure correct operation.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

9.2 The Translation Lookaside Buffer

The *Translation Lookaside Buffer* (TLB) is a cache of recently executed page translations within the MMU. On a memory access, the MMU first checks whether the translation is cached in the TLB. If the requested translation is available, you have a TLB hit, and the TLB provides the translation of the physical address immediately. If the TLB does not have a valid translation for that address, you have a TLB miss and an external translation table walk is required. This newly loaded translation can then be cached in the TLB for possible reuse.

The exact structure of the TLB differs between implementations of the ARM processors. What follows is a description of a typical system, but individual implementations might vary from this. There are one or more micro-TLBs that are situated close to the instruction and data caches. Addresses with entries that hit in the micro-TLB require no additional memory look-up and no cycle penalty. However, the micro-TLB has only a small number of mappings, typically eight on the instruction side and eight on the data side. This is backed by a larger main TLB (typically 64 entries), but there might be some penalty associated with accesses that miss in the micro-TLB but that hit in the main TLB. Figure 9-3 shows how each TLB entry contains physical and virtual addresses, but also attributes (such as memory type, cache policies and access permissions) and potentially an ASID value, described in *Address Space ID* on page 9-17.

The TLB is like other caches and so has a TLB line replacement policy, but this is effectively transparent to users. If the translation table entry is a valid one, the virtual address, physical address and other attributes for the whole page or section are stored as a TLB entry. If the translation table entry is not valid, the TLB will not be updated. The ARM architecture requires that only valid translation table descriptors are cached within the TLB.

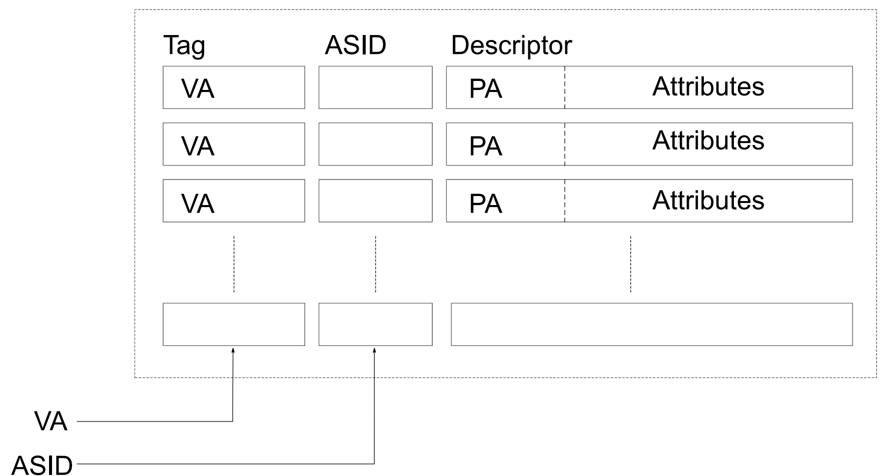

Figure 9-3 Illustration of TLB structure

9.2.1 TLB coherency

When the operating system changes translation table entries, it is possible that the TLB could contain stale translation information. The OS must take steps to invalidate TLB entries. There are several CP15 operations available that permit a global invalidate of the TLB or removal of specific entries.

As speculative instruction fetches and data reads might cause translation table walks, it is essential to invalidate the TLB when a valid translation table entry is changed. Invalid translation table entries cannot be cached in the TLB so they can be changed without invalidation.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

The Linux kernel has a number of functions that use these CP15 operations, including `flush_tlb_all()` and `flush_tlb_range()`. Such functions are not typically required by device drivers.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

9.3 Choice of page sizes

Page size is essentially controlled by the operating system, but it is worth being aware of the considerations involved when selecting a size. Smaller page sizes enable finer control of a block of memory and potentially can reduce the amount of unused memory in a page. If a task requires 7KB of data space, there is less unused space if it is allocated two 4KB pages as opposed to a 64KB page or a 1MB section. Smaller page sizes also enable finer control over permissions, cache properties and so forth.

However, with increased page sizes, each entry in the TLB holds a reference to a larger piece of memory. It is therefore more likely that a TLB hit will occur on any access and so there will be fewer translation table walks to slow external memory. For this reason, 16MB supersections can be used with large pieces of memory that do not require detailed mapping. In addition, each L2 translation table requires 1KB of memory.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

9.4 First level address translation

Consider the process by which a virtual address is translated to a physical address using level 1 translation table entries on an ARM core. The first step is to locate the translation table entry associated with the virtual address.

The first stage of translation uses a single level 1 translation table, sometimes called a master translation table. The L1 translation table divides the full 4GB address space of a 32-bit core into 4096 equally sized sections, each of which describes 1MB of virtual memory space. The L1 translation table therefore contains 4096 32-bit (word-sized) entries.

Each entry can either hold a pointer to the base address of a level 2 translation table or a translation table entry for translating a 1MB section. If the translation table entry is translating a 1MB section determined by the encoding, (See Figure 9-5 on page 9-8), it gives the base address of the 1MB page in physical memory.

The lower bits are the same in both addresses (defining an offset in physical memory from the base address). The ARM MMU supports a multi-level translation table architecture with two levels of translation tables, level 1 (L1) and level 2 (L2). Unless the Large Physical Address Extensions (See *Large Physical Address Extensions* on page 22-10) are implemented, both L1 and L2 translation tables use the *Short-descriptor* translation table format that feature:

- 32-bit page descriptors.
- Up to two levels of translation tables.
- Support for 32-bit physical addresses
- Support for the following memory sizes:
  - 16 MB or 1 MB sections.
  - 64KB or 4KB page sizes.

The base address of the L1 translation table is known as the *Translation Table Base Address* and is held in CP15 c2. It must be aligned to a 16KB boundary. The Translation table locations are defined by the *Translation Table Base Register*s (TTRB0 and TTRB1).

When the MMU performs a translation, the top 12 bits of the requested virtual address act as the index into the translation table.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

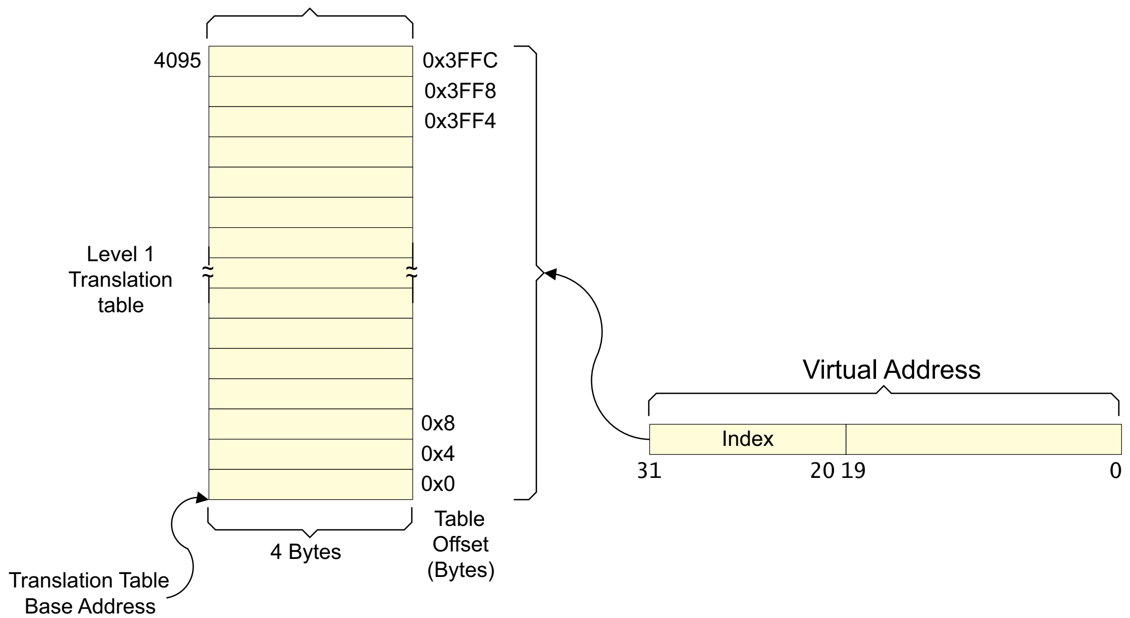

Figure 9-4 Finding the address of the level 1 translation table entry

To take a simple example, shown in Figure 9-4, suppose the L1 translation table is stored at address `0x12300000`. The processor issues virtual address `0x00100000`. The top 12 bits [31:20] define which 1MB of virtual address space is being accessed. In this case `0x001`, so the MMU must read table entry 1. To get the offset into the table you must multiply the entry number by entry size:

`0x001` * 4 bytes = address offset of `0x004`

The address of the entry the MMU reads the physical address from is `0x12300000` + `0x004` = `0x12300004`.

Now that you have the location of the translation table entry, you can use it to determine the physical memory address

Figure 9-5 shows the format of L1 translation table entries in CP15 c2.

	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
Fault	Ignored																														0	0
Pointer to 2nd level page table	Level 2 Descriptor Base Address																						P	Domain				SBZ			0	1
Section	Section Base Address												SBZ	0	nG	S	APX	TEX			AP		P	Domain				XN	C	B	1	0
Supersection	Supersection Base Address								SBZ					1	nG	S	APX	TEX			AP		P	Domain				XN	C	B	1	0

Figure 9-5 Level 1 translation table entry format

First level translation tables contain first level descriptors.

L1 translation table entries can be one of four possible types:

- A 1MB section translation entry, mapping a 1MB region to a physical address.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

- An entry that points to an L2 translation table. This enables a 1MB piece of memory to be sub-divided into pages.
- A 16MB supersection. This is a special kind of 1MB section entry, that requires 16 entries in the translation table, but can reduce the number of entries allocated in the Translation Lookaside Buffer for this region.
- A fault entry that generates an abort exception. This can be either a prefetch or data abort, depending on the type of access. This effectively indicates virtual addresses that are unmapped.

The least significant two bits [1:0] in the entry define whether the entry is a fault entry, a translation table entry, or a section entry. Bit [18] is used to distinguish between a normal section and supersection.

A supersection is a 16MB piece of memory, that must have both its virtual and physical base address aligned to a 16MB boundary. Because L1 translation table entries each describe 1MB, you require 16 consecutive, identical entries within the table to mark a supersection. *Choice of page sizes* on page 9-6 described why supersections can be useful.

Figure 9-6 shows the simplest case in which the physical address of a 1MB section is directly generated from the contents of a single entry in the level 1 translation table.

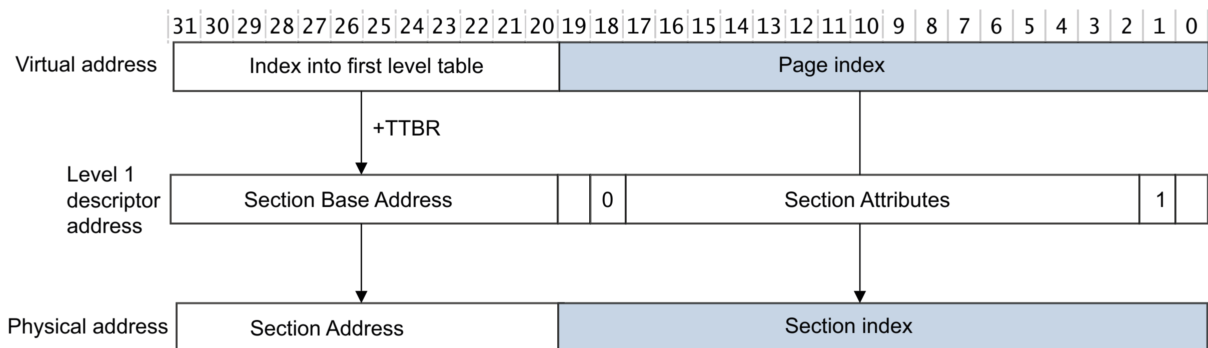

Figure 9-6 First level address translation

The translation table entry for a section (or supersection) contains the physical base address used to translate the virtual address. Many other bits are given in the translation table entry, including the Access Permissions (AP) and Cacheable (C) or Bufferable (B) types that we will consider in *Memory attributes* on page 9-14. This is all of the information required to access the corresponding physical address and in these cases, there is no requirement for the MMU to look beyond the L1 table.

Figure 9-7 on page 9-10 summarizes the translation process for an address translated by a section entry in the L1 translation table.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

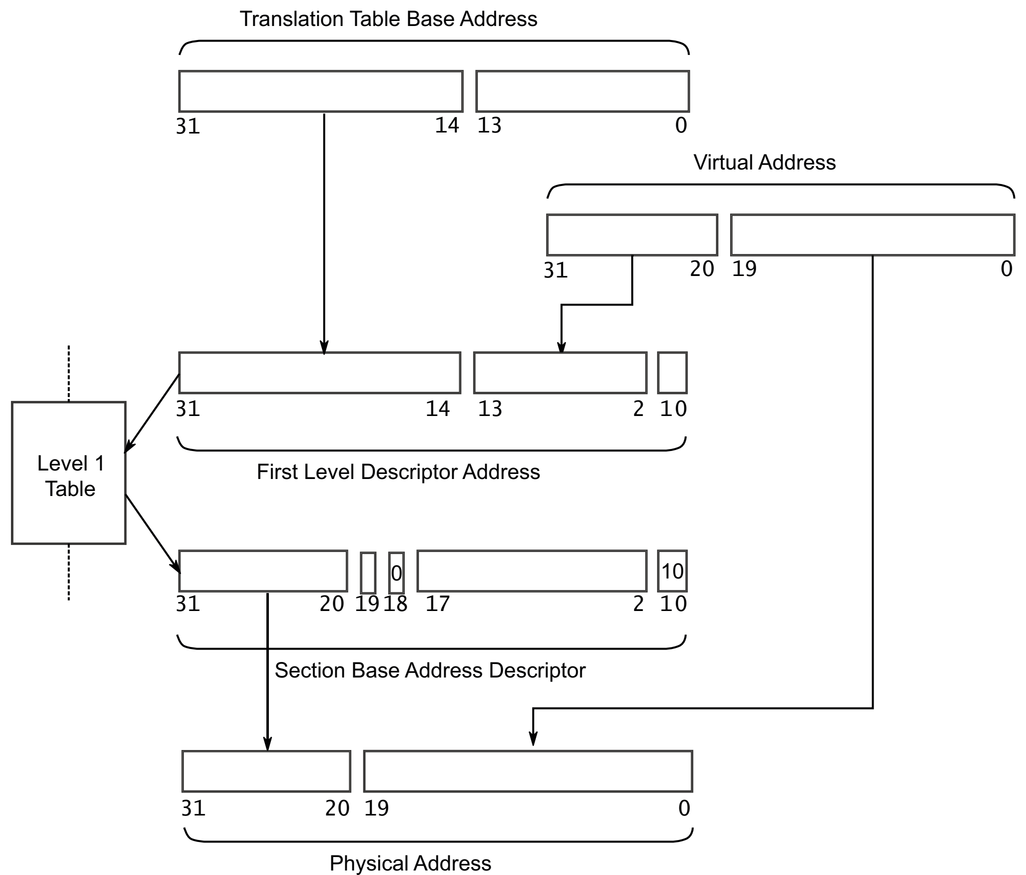

Figure 9-7 Generating a physical address from a level 1 translation table entry

In a translation table entry for a 1MB section of memory, the upper 12 bits of the translation table entry replace the upper 12 bits of the virtual address when generating the physical address, as Figure 9-5 on page 9-8 shows.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

9.5 Level 2 translation tables

An L2 translation table has 256 word-sized (4 byte) entries, requires 1KB of memory space and must be aligned to a 1KB boundary. Each entry translates a 4KB block of virtual memory to a 4KB block in physical memory. A translation table entry can give the base address of either a 4KB or 64KB page.

There are three types of entry used in L2 translation tables, identified by the value in the two least significant bits of the entry:

- A large page entry points to a 64KB page. You should note that, since each entry points to a 4KB address space, large page entries must be repeated 16 times.
- A small page entry points a 4KB page.
- A fault page entry generates an abort exception if accessed.

Figure 9-8 shows the format of L2 translation table entries.

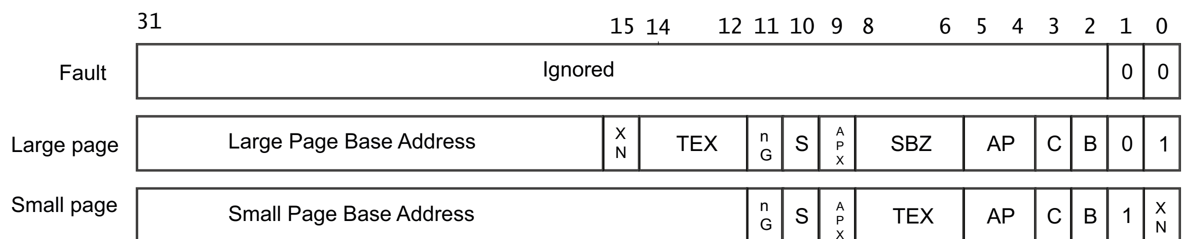

Figure 9-8 Format of a level 2 translation table entry

As with the L1 translation table entry, a physical address is given, along with other information about the page. Type extension (TEX), Shareable (S), and Access Permission (AP, APX) bits are used to specify the attributes necessary for the ARMv7 memory model. Along with TEX, the C and B bits control the cache policies for the memory governed by the translation table entry. The nG bit defines the page as being global (applies to all processes) or non-global (used by a specific process). These are described in more detail in *Memory attributes* on page 9-14.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

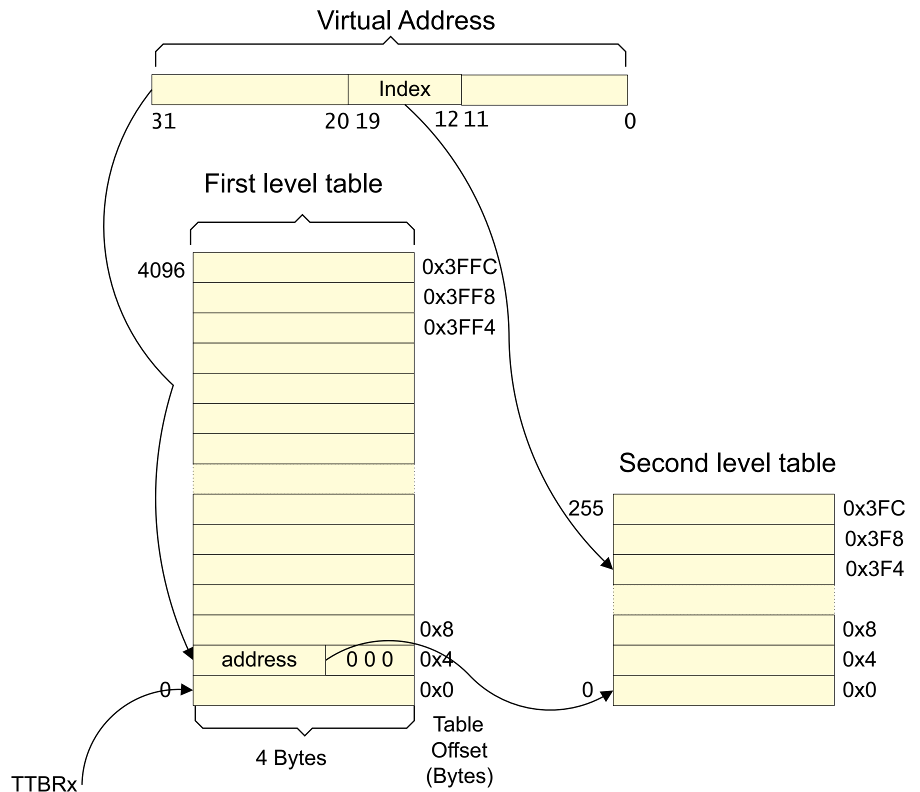

Figure 9-9 Generating the address of the level 2 translation table entry

In Figure 9-9 we see how the address of the L2 translation table entry that we require is calculated by taking the (1KB aligned) base address of the level 2 translation table (given by the level 1 translation table entry) and using 8 bits of the virtual address (bits [19:12]) to index within the 256 entries in the L2 translation table.

Figure 9-10 on page 9-13 summarizes the address translation process when using two layers of translation tables. Bits [31:20] of the virtual address are used to index into the 4096-entry L1 translation table, whose base address is given by the CP15 TTB register. The L1 translation table entry points to an L2 translation table that contains 256 entries. Bits [19:12] of the virtual address are used to select one of those entries that then gives the base address of the page. The final physical address is generated by combining that base address with the remaining bits of the virtual address.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

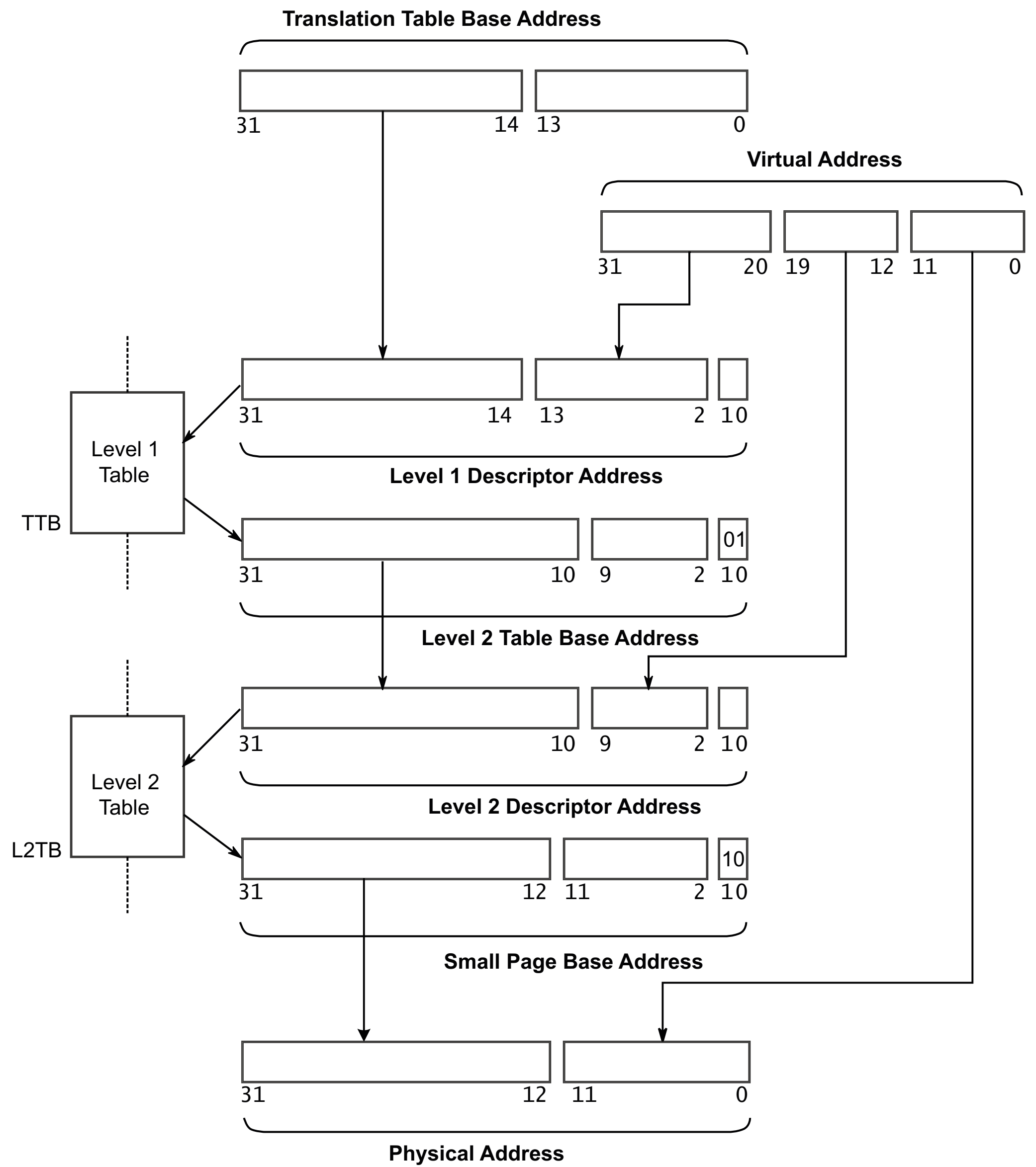

Figure 9-10 Summary of generation of physical address using the L2 translation table entry

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

9.6 Memory attributes

We have seen how translation table entries enable the MMU hardware to translate virtual to physical addresses. However, they also specify a number of attributes associated with each page, including access permissions, memory type and cache policies.

9.6.1 Memory Access Permissions

The Access Permission (AP and APX) bits in the translation table entry give the access permissions for a page. See Table 9-1.

An access that does not have the necessary permission (or that faults) will be aborted. On a data access, this will result in a precise data abort exception. On an instruction fetch, the access will be marked as aborted and if the instruction is not subsequently flushed before execution, a prefetch abort exception will be taken. Faults generated by an external access will not, in general, be precise.

Information about the address of the faulting location and the reason for the fault is stored in CP15 (the fault address and fault status registers). The abort handler can then take appropriate action – for example, modifying translation tables to remedy the problem and then returning to the application to retry the access. Alternatively, the application that generated the abort might have a problem and must be terminated.

Table 9-1 Summary of Access Permission encodings

APX	AP	Privileged	Unprivileged	Description
0	00	No access	No access	Permission fault
0	01	Read/Write	No access	Privileged Access only
0	10	Read/Write	Read	No user-mode write
0	11	Read/Write	Read/Write	Full access
1	00	-	-	Reserved
1	01	Read	No access	Privileged Read only
1	10	Read	Read	Read only
1	11	-	-	Reserved

9.6.2 Memory types

Earlier ARM architecture versions enabled you to specify the memory access behavior of pages by configuring whether the cache and write buffer could be used for that location. This simple scheme is inadequate for today's more complex systems and processors, where you can have multiple levels of caches, hardware managed coherency between multiple processors sharing memory and processors that can speculatively fetch both instructions and data. The new memory types added to the ARM architecture in ARMv6 and extended in the ARMv7 architecture are designed to meet these requirements.

Three mutually exclusive memory types are defined in the ARM architecture. All regions of memory are configured as one of these three types:

- Strongly-ordered
- Device
- Normal.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

These are used to describe the memory regions. A summary of the memory types is shown in Table 9-2.

Table 9-2 Memory attributes

Memory type	Shareable/ Non-shareable	Cacheable	Description
Normal	Shareable	Yes	Designed to handle normal memory that is shared between multiple cores.
	Non-shareable	Yes	Designed to handle normal memory that is used only by a single core.
Device	-	No	Designed to handle memory-mapped peripherals.[a] All memory accesses to Device memory occur in program order.
Strongly-ordered	-	No	All memory accesses to Strongly-ordered memory occur in program order. All Strongly-ordered accesses are assumed to be shared.

a. Shared memory was originally used to distinguish between accesses directed to the "peripheral private port" found on several ARM11 processors. This use is now deprecated and processors implementing LPAE treat all device accesses as Shareable.

More descriptions of these memory types can be found in *ARM memory ordering model* on page 10-3.

Table 9-3 shows how the TEX, C and B bits within the translation table entry are used to set the memory types of a page and also the cache policies to be used. The meaning of each of the memory types is described in Chapter 10, while the cache policies were described in Chapter 8.

Table 9-3 Memory type and cacheable properties encoding in translation table entry

TEX	C	B	Description	Memory type
000	0	0	Strongly-ordered	Strongly-ordered
000	0	1	Shareable device	Device[a]
000	1	0	Outer and Inner write-through, no allocate on write	Normal
000	1	1	Outer and Inner write-back, no allocate on write	Normal
001	0	0	Outer and Inner non-cacheable	Normal
001	-	-	Reserved	-
010	0	0	Non-shareable device	Device[a]
010	-	-	Reserved	-
011	-	-	Reserved	-
1XX	Y	Y	Cached memory XX = Outer policy YY = Inner policy	Normal

a. LPAE treats all device accesses as Shareable

The final entry within the table requires more explanation. For normal cacheable memory, the two least significant bits of the TEX field are used to provide the outer cache policy (perhaps for level 2 or level 3 caches) while the C and B bits give the inner cache policy (for level 1 and

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

any other cache that is to be treated as inner cache). This enables you to specify different cache policies for both the inner and outer cache. For the Cortex-A15 and Cortex-A8 processors, inner cache properties set by the translation table entry apply to both L1 and L2 caches. On some older processors, outer cache might support write allocate, while the L1 cache might not. Such processors must still behave correctly when running code that requests this cache policy, of course.

9.6.3 Execute Never

When set, the *Execute Never* (XN) bit in the translation table entry prevents speculative instruction fetches taking place from desired memory locations and will cause a prefetch abort to occur if execution from the memory location is attempted. Typically device memory regions are marked as execute never to prevent accidental execution from such locations, and to prevent undesirable side-effects which might be caused by speculative instruction fetches.

9.6.4 Domains

The ARM architecture has an unusual feature that enables regions of memory to be tagged with a domain ID. There are 16 domain IDs provided by the hardware and CP15 c3 contains the *Domain Access Control Register* (DACR) that holds a set of 2-bit permissions for each domain number. This enables each domain to be marked as no-access, *manager* mode or *client* mode. No-access causes an abort on any access to a page in this domain, irrespective of page permissions. Manager mode ignores all page permissions and enables full access. Client mode uses the permissions of the pages tagged with the domain.

Note

The use of domains is deprecated in the ARMv7 architecture, and will eventually be removed, but in order for access permissions to be enforced, it is still necessary to assign a domain number to a section and to ensure that the permission bits for that domain are set to client. Typically, you would set all domain ID fields to 0 and set all fields in the DACR to ‘Client’.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

9.7 Multi-tasking and OS usage of translation tables

In most systems using Cortex-A series processors, you will have a number of applications or tasks running concurrently. Each task can have its own unique translation tables residing in physical memory. Typically, much of the memory system is organized so that the virtual-to-physical address mapping is fixed, with translation table entries that never change. This typically is used to contain operating system code and data, and also the translation tables used by individual tasks.

Whenever an application is started, the operating system will allocate it a set of translation table entries that map both the code and data used by the application to physical memory. If the application has to map in code or extra data space (for example through a `malloc()` call), the kernel can subsequently modify these tables. When a task completes and the application is no longer running, the kernel can remove any associated translation table entries and re-use the space for a new application. In this way, multiple tasks can be resident in physical memory. On a task switch, the kernel switches translation table entries to the page in the next thread to be run. In addition, the dormant tasks are completely protected from the running task. This means that the MMU can prevent the running task from accessing the code or data of other tasks.

9.7.1 Address Space ID

When we described the translation table bits in *Level 2 translation tables* on page 9-11 we noted a bit called nG (non-global). If the nG bit is set for a particular page, the page is associated with a specific application. When the MMU performs a translation, it uses both the virtual address and an ASID value.

The ASID is a number assigned by the OS to each individual task. This value is in the range 0-255 and the value for the current task is written in the ASID register (accessed using CP15 c13). When the TLB is updated and the entry is marked as non-global, the ASID value will be stored in the TLB entry in addition to the normal translation information. Subsequent TLB look-ups will only match on that entry if the current ASID matches with the ASID that is stored in the entry. You can therefore have multiple valid TLB entries for a particular page (marked as non-global), but with different ASID values. This significantly reduces the software overhead of *context switches*, as it avoids the requirement to flush the on-chip TLBs. The ASID forms part of a larger (32-bit) process ID register that can be used in task-aware debugging.

Note

A *context switch* denotes the scheduler transferring execution from one process to another. This typically requires saving the current process state and restoring the state of the next process waiting to be run.

Figure 9-11 on page 9-18 illustrates this. Here, you have multiple applications (A, B and C), each of which is linked to run from virtual address 0. Each application is located in a separate address space in physical memory. There is an ASID value associated with each application so you can have multiple entries within the TLB at any particular time, that will be valid for virtual address 0.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

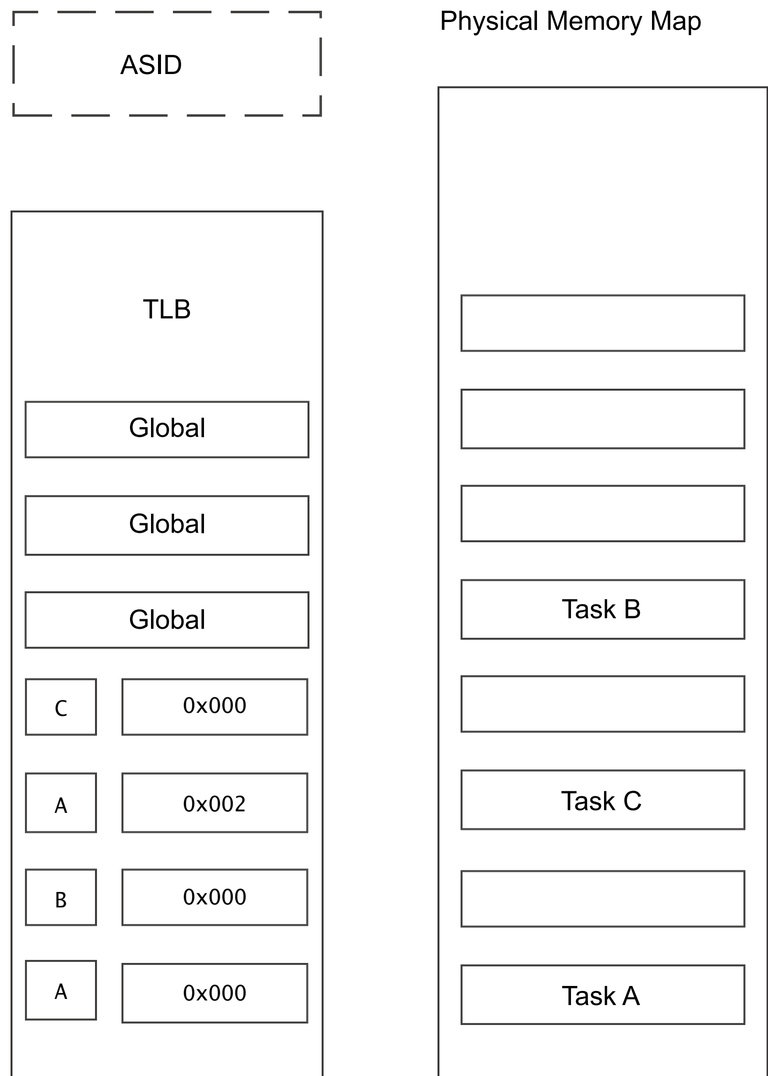

Figure 9-11 ASIDs in TLB mapping the same virtual address

9.7.2 Translation Table Base Register 0 and 1

An additional potential difficulty associated with managing multiple applications with their individual translation tables is that there could be multiple copies of the L1 translation table, one for each application. Each of these will be 16KB in size. Most of the entries will be identical in each of the tables, as typically only one region of memory will be task-specific, with the kernel space being unchanged in each case. Furthermore, if a global translation table entry is to be modified, the change will be required in each of the tables.

To help reduce the effect of these problems, a second translation table base register is provided. CP15 contains two *Translation Table Base Registers*, TTBR0 and TTBR1. A control register (the TTB Control Register) is used to program a value in the range 0 to 7. This value (denoted by N) tells the MMU how many of the upper bits of the virtual address it must check to determine which of the two TTB registers to use.

When N is 0 (the default), all virtual addresses are mapped using TTBR0. With N in the range 1-7, the hardware looks at the most significant bits of the virtual address. If the N most significant bits are all zero, TTBR0 is used, otherwise TTBR1 is used.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

For example, if N is set to 7, any address in the bottom 32MB of memory will use TTBR0 and the rest of memory will use TTBR1. As a result, the application-specific translation table pointed to by TTBR0 will contain only 32 entries (128 bytes). The global mappings are in the table pointed to by TTBR1 and only one table must be maintained.

When these features are used, a context switch will typically require the operating system to change the TTBR0 and ASID values, using CP15 instructions. However, as these are two separate, non-atomic operations, some care is required to avoid problems associated with speculative accesses occurring using the new value of one register together with the older value of the other. OS programmers making use of these features should become familiar with the sequences recommended for this purpose in the *ARM Architecture Reference Manual*.

9.7.3 The Fast Context Switch Extension

The *Fast Context Switch Extension* (FCSE) was added to the ARMv4 architecture but has been deprecated since ARMv6. It enabled multiple independent tasks to run in a fixed, overlapping area at the bottom of the virtual memory space without having to clean the cache or TLB on a context switch. It did this by modifying virtual addresses by substituting a process ID value into the top seven bits of the virtual address (but only if that address lay within the bottom 32MB of memory). Some ARM documentation distinguishes *Modified Virtual Addresses* (MVA) from *Virtual Addresses* (VA). This distinction is useful only when the FCSE is used.

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential