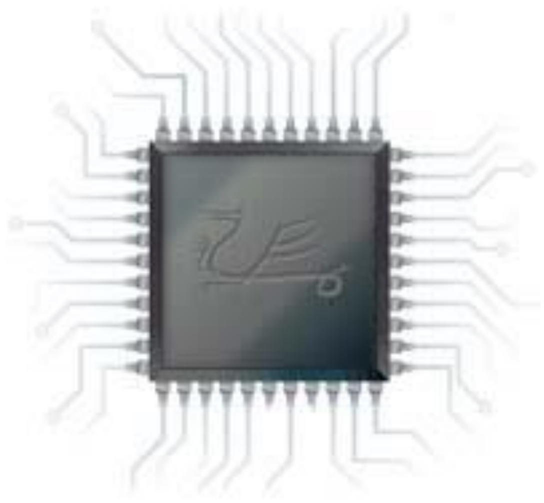
Demystifying Memory Sub-systems



Simon Southwell

July 2022

Preface

This document brings together two articles written in July 2022, and published on LinkedIn, that cover memory sub-systems, with caches, virtual memory, memory protection and MMUs. The articles discuss practical architectures and give real-world examples, such as the virtual memory support defined in the RISC-V specifications.

Simon Southwell
Cambridge, UK
July 2022

Contents

Part 1: Caches

Introduction

In this and the next article I want to cover the topic of memory sub-systems, discussing subjects such as caches, memory protection, virtual memory, and memory-management systems. It is my experience over the years that some engineers seem to think these topics are too difficult and specialist to understand—even 'magic'. This depends where one finds oneself working, of course, and I have worked in environments where these functions need to be implemented and expertise is at hand. So in these articles I want to demystify these topics and show that, even if this is not an area of expertise or experience it is not really a difficult subject if approached at the right pace and appreciate the context of what problems are being solved by these systems. I also think that many logic and embedded software engineers will at least be working within a processor based SoC system with these type of memory sub-systems, such as an ARM or RISC-V based SoC and knowing how these systems work is extremely helpful in understanding how to use them effectively, and to debug issues when constructing a solution based on them. This first article, though, will concentrate on caches, what types are common and why they are employed.

A History Lesson

In 1981, at the early stages of the personal computer revolution, several PC computers were launched; the Sinclair ZX81, The BBC Micro, the Commodore 64 and the IBM PC, amongst others. These systems had certain system attributes in common, such as an 8-bit microprocessor, RAM, ROM and peripherals for keyboard, sound, graphics, tape/disk interface and programmable input/output. The operating system might be stored on the ROM and programs loaded into RAM. When the computer started running it would read instructions over its memory mapped bus directly from ROM and/or RAM. Memory was expensive and these early systems were limited to tens of kilobytes (or less—the ZX81 was shipped with 1Kbyte). The dynamic random-access memory (DRAM) used at the time had comparable access times to the CPU clock rate and the CPU was not stalled on instruction accesses. In some cases, where video RAM was shared with the CPU it might even be faster. The BBC micro, for example, had a 2MHz CPU, with RAM running at 4MHz, interleaving video and CPU access, without stalling either.

As time progressed and CPUs became faster, accesses to memory could not keep up, particularly with the introduction of synchronous DRAM (SDRAM). A CPU system can function perfectly well with the slower memory, but each load or store between the CPU and memory introduces wait states that slows down the effective performance of the CPU. The solution was to introduce cache memory. That is a faster memory that contains fragments of the code and data currently being used or, more precisely, *likely* to be used.

Initially caches were external fast RAM, such as SRAM, but were quickly integrated into the microprocessors themselves as on chip memory.

Cache Requirements

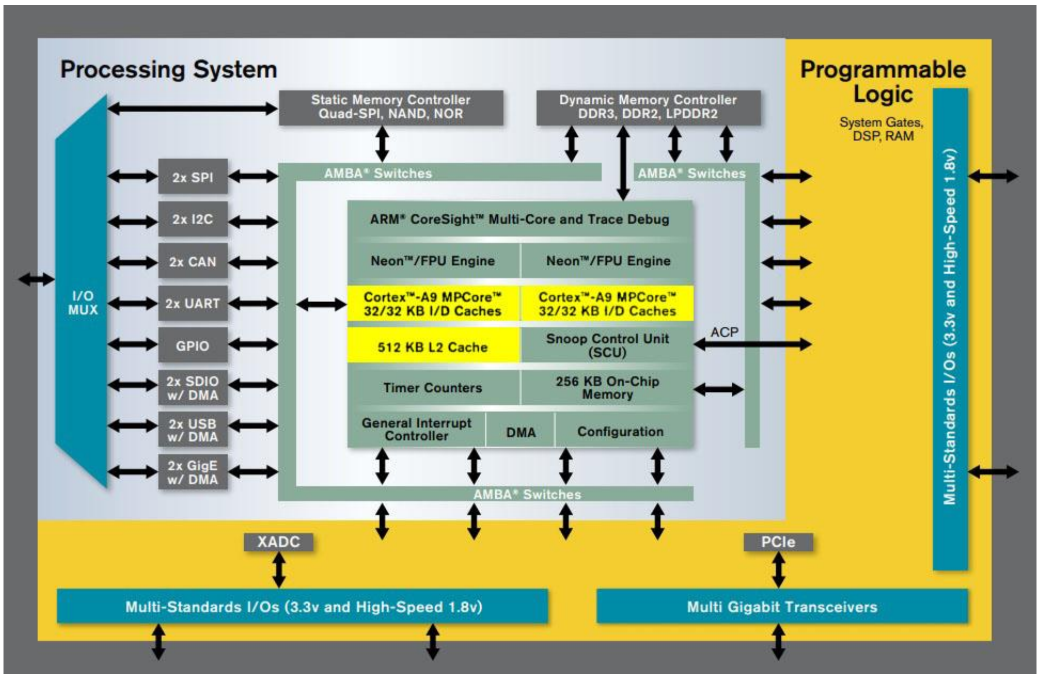
The idea behind the use of a cache makes some assumptions about the way programs tend to execute. Most programs have a linear flow for more than just a few instructions before it might branch or jump, and also mostly works on local data within those fragments of code, say on a section of memory on the stack. If a program did truly jump around at random memory locations, and access random areas of data memory a cache would not be an advantage—actually, it would slow it down.

Given this, a cache design would need to access sections of instructions or data from main memory to place into the cache memory. This is actually efficient from an SDRAM access point of view as well. Since, in modern CPU systems, multiple processes are running, it will need to keep several disparate fragments in the cache, and know which memory fragments these are (i.e. their address) and also, given the finite size of cache memory that will be available, it will need some mechanism for writing back updated fragments to main memory, in the correct place, so it can use that part of the cache memory for a new fragment, as needed.

As we shall see, the way caches work means that even they may take multiple CPU clock cycles to access, though less than accesses to main memory. In ARM (and I assume other) systems there may be some tightly coupled memory (TCM) which would be single cycle memory, holding important, time critical, code that needs fast execution, such as initial interrupt routines or special operating system code and data. TCM, however, is not to be confused with cache memory. TCM would sit at some fixed place in the memory map and would not be cached.

Caches can also multi-layered. The CPU may access memory via a cache (let's say level 1) which is very fast, but this limits the amount that can be deployed. There is a (rough) inverse relationship between RAM size and speed. So many systems will have another layer of cache (layer 2) underneath layer 1, which is larger but slower—though still faster than direct access to main memory. If data is not in the L1 cache, L2 is inspected and L1 updated if present, else main memory is accessed. System can even have a third layer (L3).

Many modern processors employ a Harvard architecture. That is, there are separate busses for instructions and data. Many systems reflect this architecture by having separate instruction and data level 1 caches, though level 2 would be common. Below is a block diagram of the Xilinx Zynq-7000 SoC FPGA.


Processing System
Programmable Logic
System Gates, DSP, RAM
Static Memory Controller Quad-SPI, NAND, NOR
Dynamic Memory Controller DDR3, DDR2, LPDDR2
AMBA® Switches
AMBA® Switches
I/O MUX
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO w/ DMA
2x USB w/ DMA
2x GigE w/ DMA
ARM® CoreSight™ Multi-Core and Trace Debug
Neon™/FPU Engine
Neon™/FPU Engine
Cortex™-A9 MPCore™ 32/32 KB I/D Caches
Cortex™-A9 MPCore™ 32/32 KB I/D Caches
512 KB L2 Cache
Snoop Control Unit (SCU)
ACP
Timer Counters
256 KB On-Chip Memory
General Interrupt Controller
DMA
Configuration
AMBA® Switches
XADC
PCIe
Multi-Standards I/Os (3.3v and High-Speed 1.8v)
Multi Gigabit Transceivers
Multi-Standards I/Os (3.3v and High-Speed 1.8v)

Highlighted in yellow, at the centre of the diagram, are the blocks for the ARM Cortex-A9 processors, indicating 32Kbyte instruction and data L1 caches, with a common 512 Kbyte L2 caches. This is mirrored in other similar devices, such as the Intel Cyclone V FPGA and gives an idea of a typical SoC setup.

So, after this overview, let’s move on to how a cache is constructed.

Cache Terminology

There is a lot of potentially confusing terminology surrounding caches, but it is all for good reason. We need to define this terminology so that we use a common, understood language.

A cache that stored individual bytes, or even words of the basic architecture size (e.g., 32 bits) as the size of fragment mention before, would not work and does not fit with the assumption about instructions and data being accessed from locally contiguous memory sections. What the size of these fragments needs to be is part of the design of the cache and is a trade-off between efficiency when accessing a block from main memory, the cache size, and the likelihood of reading values that are not accessed before the data is removed from the cache once more. This fragment of the chosen size is called a *cache line*. Typical sizes might range from 8 to 256 bytes for an L1 cache (in powers of 2). The cache line fragments would be aligned in memory to its size so that, for example, 32-byte cache line fragments will have start addresses at multiples of 32 bytes—0, 32, 64 etc.

Obviously, a single cache line in memory would not be very efficient, as it would quickly need to be replaced with a new line, with all the delay that that would cause. So, the cache has a number of cache line spaces, and the collection of cache-lines is called a *set*. A typical set size might range from 128 to 16K cache-lines.

A cache may also have more than one set. This is known as the number of *ways* and might range from 1 to 4. The total size of the cache, in bytes, is just these three numbers multiplied together. For example, if a cache-line size is 32 bytes, with a set size of 128 and is a 2-way cache, then this is 32 × 128 × 2 = 8Kbyte cache. Below is a table showing some real-world processor systems and their level 1 cache parameters (some of which are configurable):

Processor	Cache line	Set size	#Ways
ARM Cortex-A9	32 bytes	128, 256, 512	4
AMD K8	64 bytes	512	2
LatticeMico32	4, 8, 16 bytes	128, 256, 512	1,2

Associativity

Another new term (unfortunately) that defines how a cache will function—associativity. Basically there are two types of associativity

- Fully associative
- Set associative

What it indicates is how the particular cache-line addresses are arranged and searched for to see if an address being accessed is actually in the cache.

Fully Associative Caches

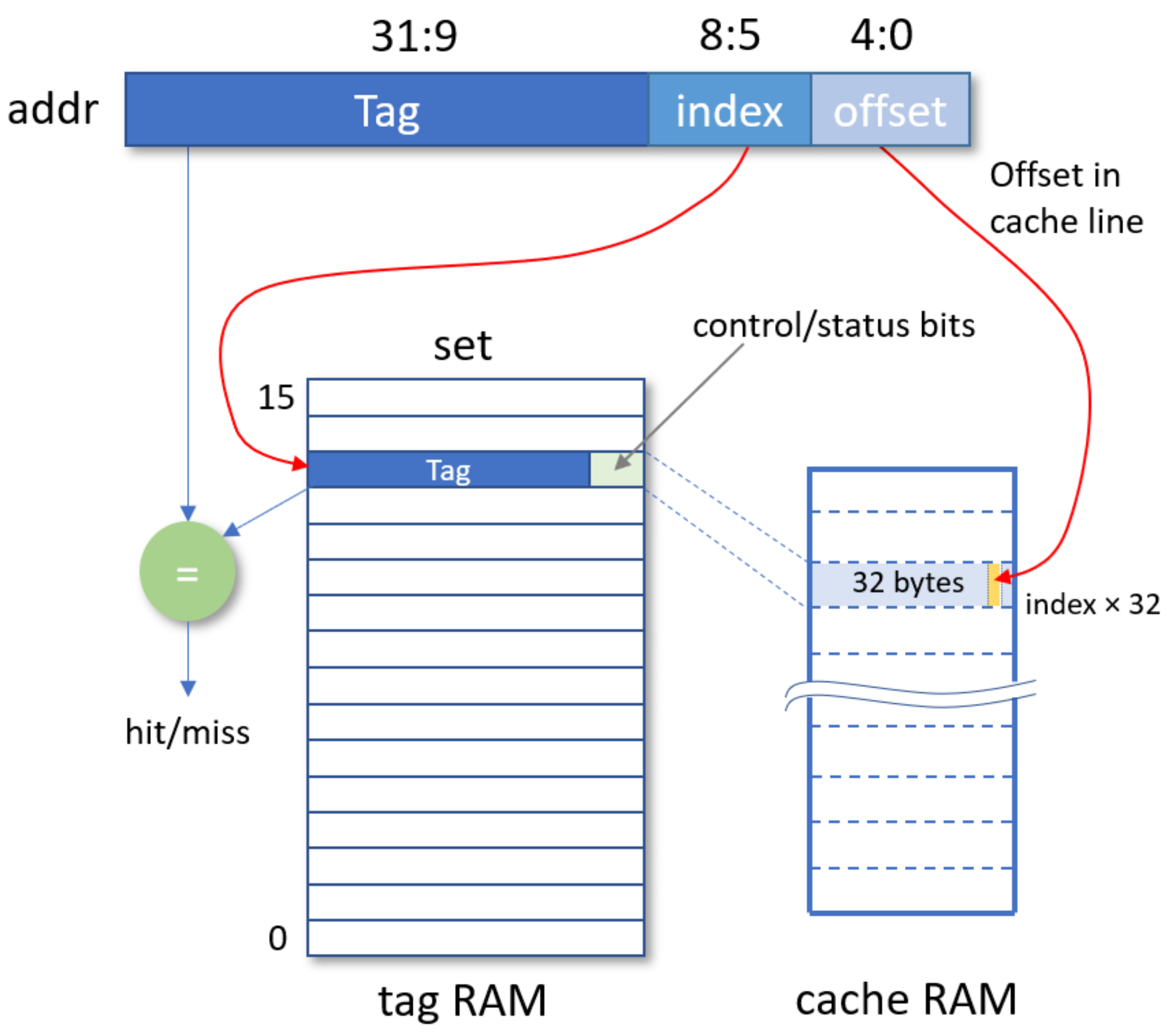
A fully associative cache is one where a cache-line can be placed anywhere within a set. The data will be placed in the cache RAM and, in order to identify where in memory that cache line resides, a 'tag' RAM will store the relevant address bits. In a fully associative cache, since this can be in any aligned location in the cache, when checking that an access is to an address in the cache, every location within the tag RAM will need to be checked to see if the access matches an address within one of the store cache-lines.

Functionally, this is the simplest to understand. One has a finite number of cache-lines spaces in a set, and any can be used to store a given cache-line. When it's full, some mechanism (more later) can be used to release one of the lines currently in use and replace with the new required cache-line. From an implementation point of view, this is an expensive mechanism. Each lookup of the tag RAM requires all entries to be inspected.

Normally this would be via a content addressable memory (CAM), sometimes known as an associative memory. These are larger and slower than standard RAMs, but are used in other applications, such as network address lookups and are sometimes employed as caches—but not too often, so I won't delve further here. Once we have looked at Set Associative caches, it should become obvious how a fully associative cache would work, just with less restrictions.

Set Associative Caches

Set associative caches are not as flexible as fully associative caches, but are much cheaper, faster, and simpler to implement and are, thus, more widespread. I mentioned, in passing, that a cache has two RAMs, the cache RAM itself, and the tag RAM—which would be a CAM for fully associative. The diagram below shows the set up for a single set for a set associative cache.



The diagram shows an example of a cache which has 32-byte cache-lines and a 16 entry set, giving a 512-byte cache RAM, and a tag ram with 16 entries of tag width bits. This is probably not a practical cache but is useful as an illustration.

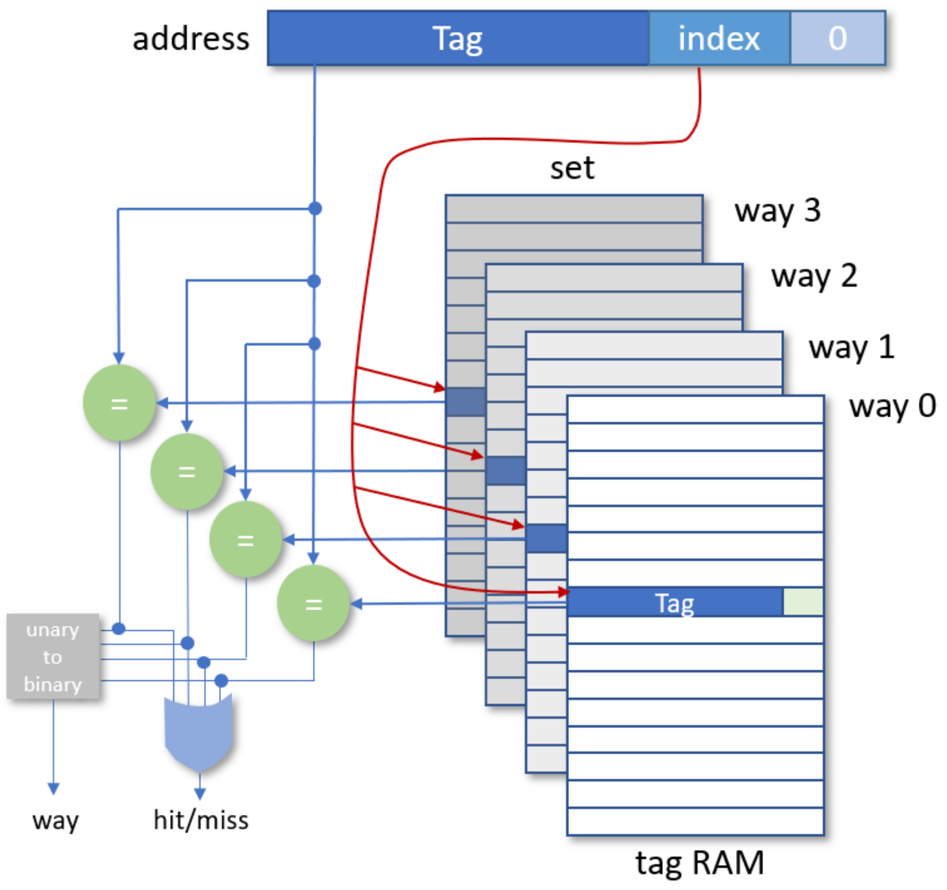
In a set associative cache, a cache-line can't be store anywhere in the cache but is a function of some bits in its address. A cache-line's address is always aligned with is size, so the bottom bits of a cache-line's address are always 0. When accessing a particular byte or word within a cache line, the access bottom address bits are an offset within the cache-line. For a 32-byte cache-line, then, the bottom 5 bits are an offset into the cache-line. The next *n* bits are used to determine which entry in the cache set the cache-line will reside. For a set of 16 entries (as per this example), this is the next 4 bits—bits 8 down to 5. All addresses that have this same value in these 'index' bits must uses that entry in the set. The remaining bits are the 'tag' and it is these bits that are store in the tag RAM, along with some control and status bits (more later).

When a processor does a load or a store to a particular address, a lookup in the tag RAM will be performed by breaking up the access address bits into the index and TAG. The index will address the entry from the tag RAM and the data read compared with the tag bits of the access address. If the values match—a 'hit'—then the addressed data is in the cache, and the address of the data in the cache RAM is the *index × cache-line size + offset*, and the data retrieved for a load, or written for a store. If the values don't match—a 'miss'—then the data is not in the cache, and the cache-line will need to be replaced with the data from main memory. The existing data in the particular cache-line may have to be written to main memory first if it has been updated without being written at the point of the miss. Once the cache-line has been replaced, then the comparison will hit, and the data read/written as for a first-time hit.

This description is for a 1 way set associative cache. This will function perfectly well but is limited in that addresses with the same index bits will clash on the same set entry, increasing the likelihood that an entry won't be in the cache and that an entry will have to be swapped. A solution is to increase the number of sets available—the 'ways'.

Multi-way Set Associative Caches

If a cache has multiple sets (i.e., the number of ways) then a cache-line address clash is reduced as it can reside in any of the ways available. As mentioned before, typical values for the number of multiple ways is 2 or 4. The diagram below extends the example by increasing the number of ways to 4.



Now the index bits can index into any of the 4 set entries in the ways. When matching a load or store address, all four tag RAMs are read at the index and all four tags compared. If any of them match, this is a hit, and the 'way' that contains the match is flagged to access the data from the cache RAM. The cache RAM might also be four different RAMs, but the way bits (2 bits for a 4-way cache) could just as easily be used as the top address bits in a larger single RAM. The choice will depend on factors like speed and resource requirements. Separate cache RAMs allow pre-fetching of data (on reads) and will be smaller and thus faster but will take up more ASIC real-estate.

If none of the ways contains a match, then this is a miss, but now an added complication arises as to which way to update? In the single set there was only one choice. Now we have to choose between *n* different ways (4 in the example). Common algorithms employed are

- Round-robin
- Least recently used (LRU)

The first is the easiest to understand and implement. For a give index one just loops round the way count each time an entry is updated with a new line: 0, 1, 2, 3, 0, 1…, and so on. This round-robin count, however, must be kept for each line in the set as there is no association between the lines and when they are updated.

In least recently used implementation a tally must be kept of when a particular way was accessed at a given index. When a miss occurs, the entry that was accessed the furthest back in time is chosen to be updated. This sound like it might be complicated, but the number of ways is not likely to be huge (4 is a typical value, as we've seen). The way I have approached this before is to assign each entry across the ways a value 0 to *n*, with 0 being the least recent access and n being the most recent. For a 4-way cache, *n* is 3. Each time there is a hit on a given index, the matching way has its LRU count updated to 3. The value it had before is used to flag that any way with an LRU value greater than this is decremented. Everything else is left unchanged. For example, if a way is accessed with previous value of 1, then the ways with values 3 and 2 are decremented to become 2 and 1, and the matched way entry set to 3. This maintains the LRU counts across the ways. Note that an LRU count must be kept for each set index, as per the round-robin method.

Cache Control and Status Bits

Along with the address TAG bits, the tag RAM will usually have addition control and status bits. As a minimum each entry will have the following bits

- Valid
- Dirty

The valid bit indicates that a valid cache-line is stored at that location. After reset (or a cache invalidation), there are no cache-line in the cache, and the valid bits are reset. When doing an address match, a valid bit that is clear forces a miss.

The dirty bit is used to indicate that a write has been performed at the cache-line entry, but that entry has not been written to main memory. When a cache line needs to be replaced, if the dirty bit is set, then the current cache-line must be written back to main memory before it can be updated. If the entry has never been written, then this write back can be skipped, saving time.

Additional, but optional, bits are often seen. Some 'accessed' bits may be present to store things such as the LRU count bits, used in choosing which line across the ways is to be replaced. Also, it may contain permission bits, indicating read, write, execute, and privilege level permissions. This is only usually done if the system does not have a separate memory protection unit, or an MMU managing these access permissions.

Write Policies

Most commonly, because it's the best performance, a cache has a write policy of 'write-back'. That is, it will write-back a cache-line to memory only when it is to be replaced and has the dirty (and valid) bit set.

Another policy used is that of write-through. Here a cache-line is written back to memory every time it is written to from the processor. This does not mean it is invalidated or replaced but means that the line is written. This increases the number of writes to main memory over a write-back policy, making it have a decreased performance. It is used where more than one processing element or other function have access to main memory and need to access data from the processor updating the cache. If a write-back policy, there can be a mismatch of data in a dirty cache-line and main memory for an indefinite amount of time. With write-through the data ends up in main memory after just the cache-line write latency.

Many caches also allow the ability to flush them or to invalidate them—something that an operating system might want to do, rather than a user application. Flushing requires that all dirty entries in the cache be written to main memory. The cache remains valid and can still have hits. If invalidated it is both flushed and all valid bits cleared so that no matches on previous cache-lines are possible.

Cache Coherency

The issue of dirty cache entries and main memory mismatches is a problem known as cache coherency. In a single processor system, when it is the sole accessor of the main memory, this is not an issue. When there's more than one device in the system with memory access, this might be a problem, if the blocks need to communicate through main memory.

We've seen how a write-through policy might solve this issue, but it has a high performance penalty. Also flushing ensures valid main memory but requires co-ordination with operating system software. Other solutions exist to address cache coherency. If you inspect the block diagram of the Xilinx Zynq-7000 from earlier, you will see a snoop control unit (SCU). Connected to it is a bus called ACP—an Accelerator Coherency Port. This allows, via the SCU, to check whether a memory access is in the cache and retrieve the data from there, or whether to access directly from memory. There is a penalty for this cache lookup, but it is smaller than for write-through, where every cache-line is written back whether it needs to be accessed from another source or not. Modern busses and interconnect protocols, such as AXI, support whether a cache coherent access is required or not, giving control to the lookup penalty for only those accesses that require it. Beyond this, for large distributed systems, the Coherent Hub Interface (CHI) interconnect uses message transactions for cache coherent accessing across many caches. These kinds of transactions were employed in supercomputers when I was working in that field and are now making their way into SoCs and chips with high processor counts to solve these types of coherency problems .

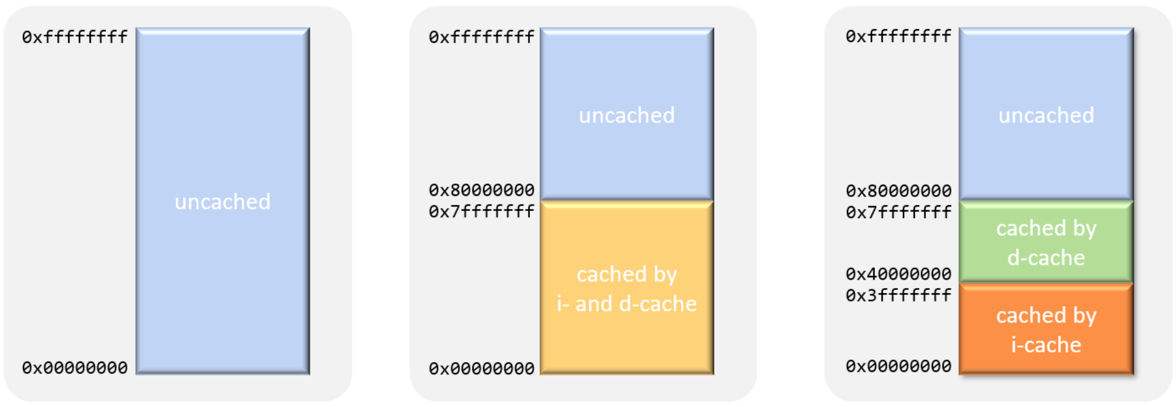
As you can see, the use of a cache solves lots of problems in slow memory accesses but create a set of new problems in coherency which must be addressed.

Cache Regions

The assumption so far is that all processor loads and stores are to go through the cache, but not all accesses to the memory map go to main memory. In an SoC, the control and status registers of peripherals may be memory mapped in the same address space. It would not be useful, for instance, to cache accesses to control registers when writing, say, a 'go' bit to start a transmission. This is controlled by defining cache regions.

At its simplest a system might mirror an address range, so that a 1 Gbyte region exists at `0x00000000` to `0x3fffffff` and is mirrored at `0x80000000` to `0xbfffffff`. The first of these regions might be cached whilst the second region, though accessing the same physical memory, would be un-cached. The top bit of the address defines whether the cache is bypassed or not.

More generally, a cache system may have the ability to define multiple regions of cached and un-cached accesses. These might be separate for instruction and data caches. The LatticeMico32 softcore processor's cache, for example, has just such definable regions. The diagram below shows some examples of the configurability of cached regions.



No cached regions

Overlapped cached regions

Separate cached regions

More sophisticated systems may allow more, and discontiguous, regions to be defined.

Conclusions

In this article was discussed how there is a mismatch on processor speed versus accesses to main memory, causing bottlenecks in programs at loads and store, and how caches can help solve some these issues. Caches can take on a couple of different forms of associativity, with advantages and disadvantages of both.

Having looked at some practical cache architectures it was seen that caches create new problems of coherency across a distributed system. This was looked at in terms of write policy and cache coherent accesses via snoop control units and an ACP.

Hopefully this has given you some insight into how caches work, and to terms that you may have heard but were unfamiliar to their meaning. It may be that you will never need to deal with caches directly, or implement new cache systems yourself, but I think understanding them can be crucial when working in systems with these functions. As a software or logic engineer, using the system efficiently with caches and knowing when coherency might be an issue and how to deal with it is useful knowledge.

In the next article we look at virtual memory, memory protection and memory management units (MMUs) that implement this.

Part 2: Virtual Memory

Introduction

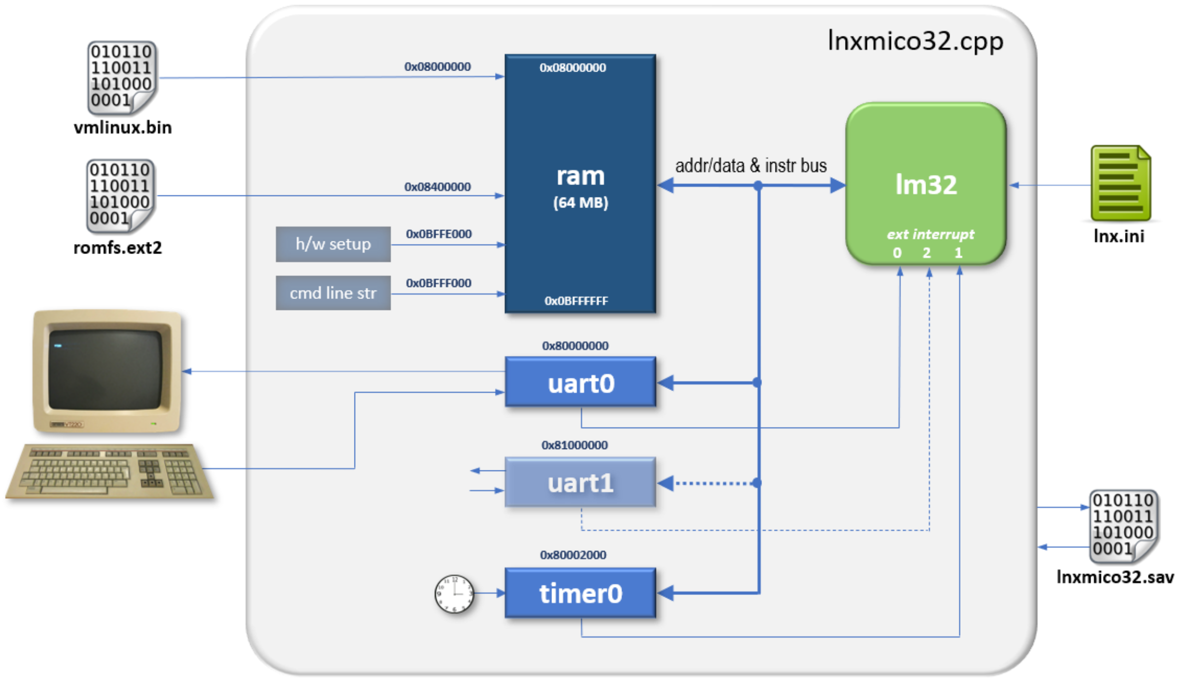
In this article we will look at virtual memory and what problems it is trying to solve in a multi-tasking environment. Dividing memory in to 'pages' to map to main memory will be discussed, as well as the tables used to keep track of these mappings. Hardware support for virtual memory is also covered in the form of memory management units (MMUs) and a real-world example of VM is discussed in the form of the RISC-V specification, using Sv32 as an example and how an MMU could use this do virtual address translation.

In the first article of this series, discussing caches, I gave a quick history lesson on what the situation was, pre-caches, in 1981 when the blossoming PC revolution was well underway. Let’s start but going back to this era which was also pre-virtual memory.

These original PCs could only perform was task at a time—they were single task machines. After reset the computer would boot and execute start up operating system code until it was ready to except user input to load some application. The OS might be in ROM (or loaded into RAM from disk or tape, via some BIOS in ROM), leaving some unreserved RAM for applications. The OS would load an application to this spare RAM and the jump to the start address of that application which would then run. The OS is no longer running in the sense that it cannot process any more user input to load and run another program. The application, of course, might make calls to OS service routines, but a single ‘thread’ of execution is maintained. When the application exits, this thread of execution returns to the OS, and new user input can be accepted to load and run other applications.

Given the speed of microprocessors and memory at the time, this was all that was possible on a PC. As CPUs become faster, and faster, larger memories became available, PCs started to have the capacity to run more than one application or task at a time. But there is a problem. The application software written for, say, DOS, would have some minimum system memory requirements to run, and would expect that memory to be at a known fixed location within the memory map. Each application would make the same assumptions about where the resources it could use would reside. To run more than one task at once would cause clashes in the use of memory.

One solution is to compile the code to only use relative addresses: so-called position-independent code (PIC). The OS can load separate applications in differing parts of memory (assuming enough memory) and run the code from their loaded locations, swapping between the tasks to give CPU time to each. (This is context switching for multi-tasking, which is outside the scope of this article.) A version of Linux, called µClinux, uses just such a method, so there is no virtual memory. As an example, my LatticeMico32 instruction set simulator (mico32) can boot this version of Linux, which also uses µClibc and BusyBox for a multi-tasking environment and user shell. The diagram below, taken from the ISS manual, shows the fairly straightforward system required to boot this OS without a memory management unit and VM support. (The ISS does actually have a basic MMU model, but it is not used in this scenario.)



This approach, though quite useful, is limiting in that all the relevant code must reside in RAM at the same time, which will quickly run out as more tasks need to be run together 'concurrently'.

A better solution might involve fooling each task into having the illusion that it is the sole process running and has access to all the memory it would have had in the original situation of a single task computer. That is, it has a 'virtual' view of memory. In reality, the task has its memory dynamically mapped to real physical memory, and each task's memory is mapped to different parts of actual memory, avoiding clashes. On top of this, in order to support multiple tasks whose total memory requirements might exceed that of the physical memory available, parts of a task's memory might be removed from physical memory and stored on a slower storage device, such as a hard disk drive. As tasks are given CPU time by the OS, this data may be reloaded to physical memory (most likely to a different physical location from the last time it was loaded). This situation was already available on main-frame computers of the time, and quickly adopted in PCs and then embedded SoC systems. Since the application code has a virtual memory view as if it were the only task running, code from the pre-virtual memory era can run without modification.

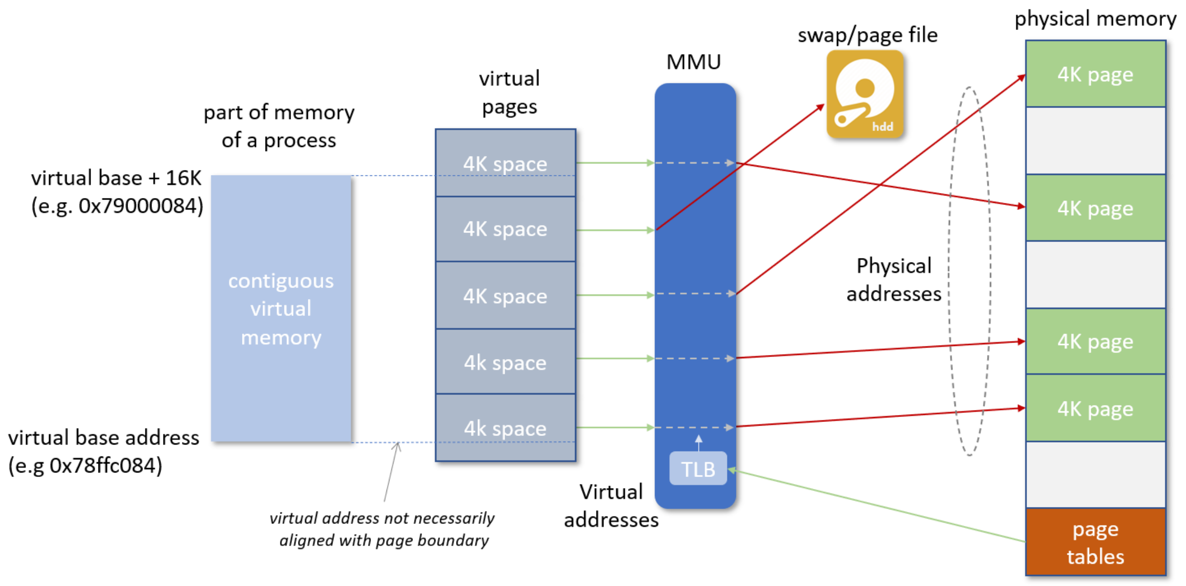
This system of memory mapping is Virtual Memory, and the in rest of this article we will discuss how one might arrange virtual memory and the operations of hardware, such as a memory management unit (MMU), to support this virtual memory in a multi-tasking system.

We will look at how memory is divided into pages, how the pages in virtual memory are mapped to physical memory and how this mapping is tracked and managed efficiently.

Pages

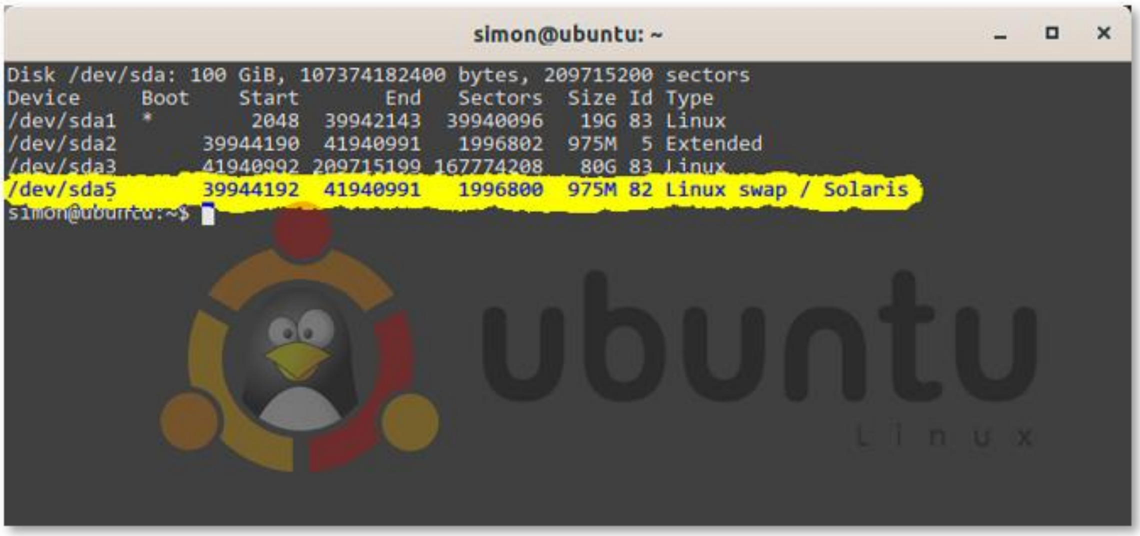
In order to get a handle on managing virtual memory, the memory is divided into ‘pages’. That is, fixed blocks of contiguous memory. A virtual memory system may support only a single page size, such as 4Kbytes, but some systems can support pages of a few different sizes. A 4Kbyte page size is common in operating systems such as Linux, for example. This is quite a small page size but was chosen some time ago and modern computers could deal with larger pages (and some do), but 4Kbyte is still widespread.

A particular task or process will have a virtual memory space that is contiguous. This space is divided up into pages, each with a *page number*. Since this is virtual memory, this is the virtual page number (VPN). The page number is really the top bits of the virtual address of the start of the page. Each page is aligned to its size, so that a 4Kbyte page will have the bottom 12 bits equal to 0 at its start address. The page number is then the remaining address bits. For a 32-bit system, for example, this is bits 31 down to 12. The virtual pages will be mapped to physical pages. Real physical memory is also divided into pages in the same way, with their own *physical* page numbers (PPN). When as task requires memory for its code or data, the virtual pages will be allocated to physical pages, and tables must be kept to remember which physical page addresses are being referenced when an access by the task is made to its virtual addresses. The diagram summaries the situation of virtual to physical mapping.


part of memory of a process
virtual base + 16K (e.g. 0x79000084)
contiguous virtual memory
virtual base address (e.g 0x78ffc084)
virtual address not necessarily aligned with page boundary
virtual pages
4K space
4K space
4K space
4k space
4k space
MMU
TLB
Virtual addresses
swap/page file
hdd
Physical addresses
physical memory
4K page
4K page
4K page
4K page
page tables

The left of the diagram shows part of a process running requiring some contiguous buffer—a large array, say. The buffer is larger than the page size but is contiguous in virtual memory, with the virtual address ranging from the buffer’s base address (`0x78ffc084` in the example) to the base plus buffer size (0x79000084). This address space will map over a set of contiguous virtual pages, though it needn't necessarily be aligned with the start and/or end of a page. The OS will allocate physical pages for each of the virtual pages referenced. This is done as whole pages, even though the used virtual memory doesn't use the start or end of the lowest and highest virtual pages' memory. The physical pages could be allocated to any PPN in physical memory and need not be contiguous.

In order to access the data in physical memory, tables must be kept with the translations from virtual pages to physical pages. The page tables reside in physical memory and, of course, don't have virtual page equivalents. As was mentioned before, the system can run out of physical pages to hold all the required data for every running process, and so some that have not been accessed for a while can be stored in secondary storage, as shown for one of the pages in the diagram. These are kept on swap partitions or page files, depending on the operating system. On Linux, for instance, there is a swap partition for storing these pages. The diagram below shows a Linux system's partitions, with the swap partition highlighted.



```
Disk /dev/sda: 100 GiB, 107374182400 bytes, 209715200 sectors
Device     Boot    Start       End   Sectors  Size Id Type
/dev/sda1  *        2048  39942143  39940096   19G 83 Linux
/dev/sda2       39944190  41940991   1996802  975M  5 Extended
/dev/sda3       41940992 209715199 167774208   80G 83 Linux
/dev/sda5       39944192  41940991   1996800  975M 82 Linux swap / Solaris
simon@ubuntu:~$ 
```

The pages that have been 'swapped' out to disk do not have a mapping to physical memory, and no entry in the page tables. If a swapped-out page is referenced, the lookup in the page table fails and the swap space searched. If found, then a resident page in memory is selected for removal to swap space, and the referenced page loaded to that physical page, and the page tables updated to map the virtual address of the loaded page to the selected physical page. Of course, if the virtual page is not in the swap space either, then an exception is raised. The details of how an OS does this swapping and the format of a swap partition is beyond the scope of this article, which is looking at VM from an MMU point of view. For those that want a deeper dive into OS VM functionality, I have found Mel Gorman's book *Understanding The Linux Virtual Memory Manager* helpful, which is freely available online. Chapter 11 discusses swap management.

Note that there are separate page tables for each process that is running, since each process can reference the same virtual address as another process, mapped to different physical pages in memory. That clash, of course, is where we started and why we need virtual memory. Some systems (that I have worked on) combine the page tables for all processes, with additional information to uniquely identify the process with an ID (PID) or context number. A lookup then has to match both a page number and a context number/PID.

The logic unit that manages the translations from virtual to physical pages, along with some other functionality we will discuss, is the memory management unit (MMU).

Memory Management Unit

Hardware support for virtual memory comes in the form of a memory management unit. It is usually fairly autonomous as a function in regards translating between virtual and physical pages, which needs to be done at high speed in order for the system to perform efficiently. The operating system, however, would usually be looking after the physical page allocations and the context switches, and would configure the MMU accordingly. This might be as simple as pointing the MMU to a different set of page tables when there is a context switch. The MMU would access, for a given process, a set of translations from memory, kept in a *page table*, in order to do the translations. As we shall see, it might also keep a local cache of some translation to speed up this process. The MMU would also normally be responsible for memory access protection and privilege.

Fundamentally, then, in co-ordination with the operating system, the MMU does the following things:

- Translates virtual addresses to physical addresses
- Reads these translations from page tables in memory and keeps a cache of them
- Does memory protection and monitors access privileges.

TLB

As mentioned above, page tables are kept in physical RAM and each time the processor does a load or a store to virtual memory the MMU has to translate this to a physical load or store. If it had to access the pages tables in main memory every time, then this access latency would be added to each and every load/store that the processor did. To alleviate this penalty, a cache of translations is kept in a *translation lookaside buffer* (TLB). This acts exactly like a cache for instructions or data that was discussed in the previous article. Only, now, instead of being data or instructions in the cache it is virtual to physical page translations. The logic is that same as for a data cache, just with parameters that fit translations. For instance, an entry in the page table (a Page Table Entry or PTE) for a 32-bit system might be 4 bytes, so the TLB's cache-line size would be 4 bytes. The set size might be smaller and may even be a fully associative cache but, in general, the function is the same as that for a data cache.

Page Tables

The entries in the page tables (PTEs) in RAM could all be kept in a single flat table, but this would be a problem. Searching through a table if the entries where just randomly scattered in a variable sized table would be too slow. Therefore, tables are organized where the position in the table is a function of the virtual page number. This is similar to the index bits of an address in a set associative cache, as discussed in the previous article. A flat table using this method, though, would require quite large tables. Even in a 32-bit system that can address up to 4Gbytes of memory this requires, if using 4Kbyte pages, 1 mega table entries for each possible virtual page. And this is for *every* process running. For a 64-bit system this rises to impractical values.

The solution is to have a hierarchy of page tables where the virtual page number is divided into sections, with the top bits used to index into a first table. If the PTEs are 4 bytes, then the top 10 bits might be used, and the table fit neatly into a 4Kbye page. The PTE there, instead of being a translation, is actually a pointer to another table. The next (10) bits are used to index into that table, and so on, until a translation entry is found. The number of tables traversed depends on the architectural size. For example, a 32-bit system might have just two steps, whilst a 64-bit system might have a 3 or even 4 steps. Traversing the tables like this is called a *table walk*. This could be done by the operating system and the TLB updated by software but, more efficiently, this would be done in logic with a *table walk engine* as part of the MMU functionality. The process of reading an entry and either continuing to another table, or fetching the translation is something well suited to logic. This also gives a mechanism whereby different sized pages can be supported. If a table walk terminates early by finding a translation in a higher table, then that page is $2^n \times$ *number-early-steps* times bigger than the fundamental page size. For example, with 4 Kbyte pages, and 10-bit page number sub-indexes, terminating one step before the 4Kbyte translation table gives a page size of $2^{10} \times$ 4Kbyte = 4Mbyte page.

Real World Example, RISC-V

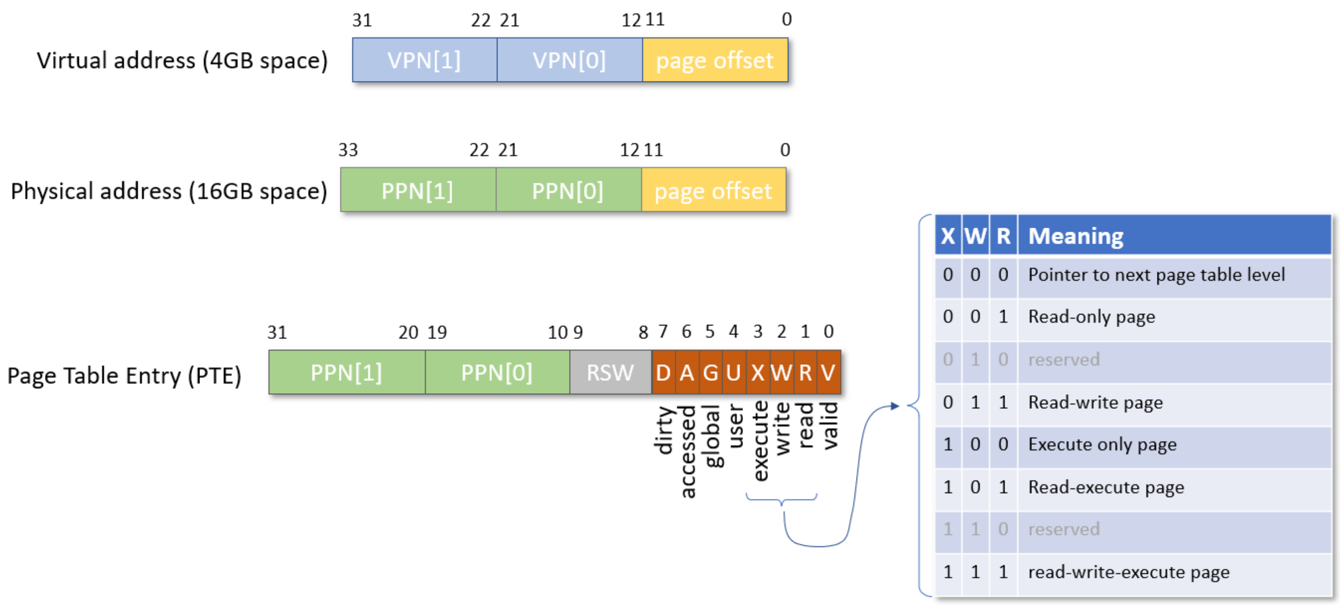
Having described the virtual memory functions of an MMU, let’s look at a real-world example to bring the theory together.

Volume 2 of the RISC-V specification (see sections 4.3 to 4.5), defines page-based support for virtual memory for various size address spaces. It defines 3 sizes, as listed below:

- Sv32 maps 32-bit virtual addresses to 34-bit physical addresses
    - 4Gbyte virtual space, 16Gbyte physical space
    - Two level page tables

- Sv39 maps 39-bit virtual addresses to 56-bit physical addresses
  - 512Gbyte virtual space, 64Pbyte physical space
  - Three level tables
- Sv48 maps 48-bit virtual addresses to 56-bit physical addresses
  - 256Tbyte virtual space, 64Pbyte physical space
  - Four level tables

We will concentrate on Sv32, as an example, as the diagrams are less messy, but I hope it will be clear how this would scale to the larger address spaces. The diagram below shows the format of the virtual- and physical addresses, along with the format of a page table entry:



X	W	R	Meaning
0	0	0	Pointer to next page table level
0	0	1	Read-only page
0	1	0	reserved
0	1	1	Read-write page
1	0	0	Execute only page
1	0	1	Read-execute page
1	1	0	reserved
1	1	1	read-write-execute page

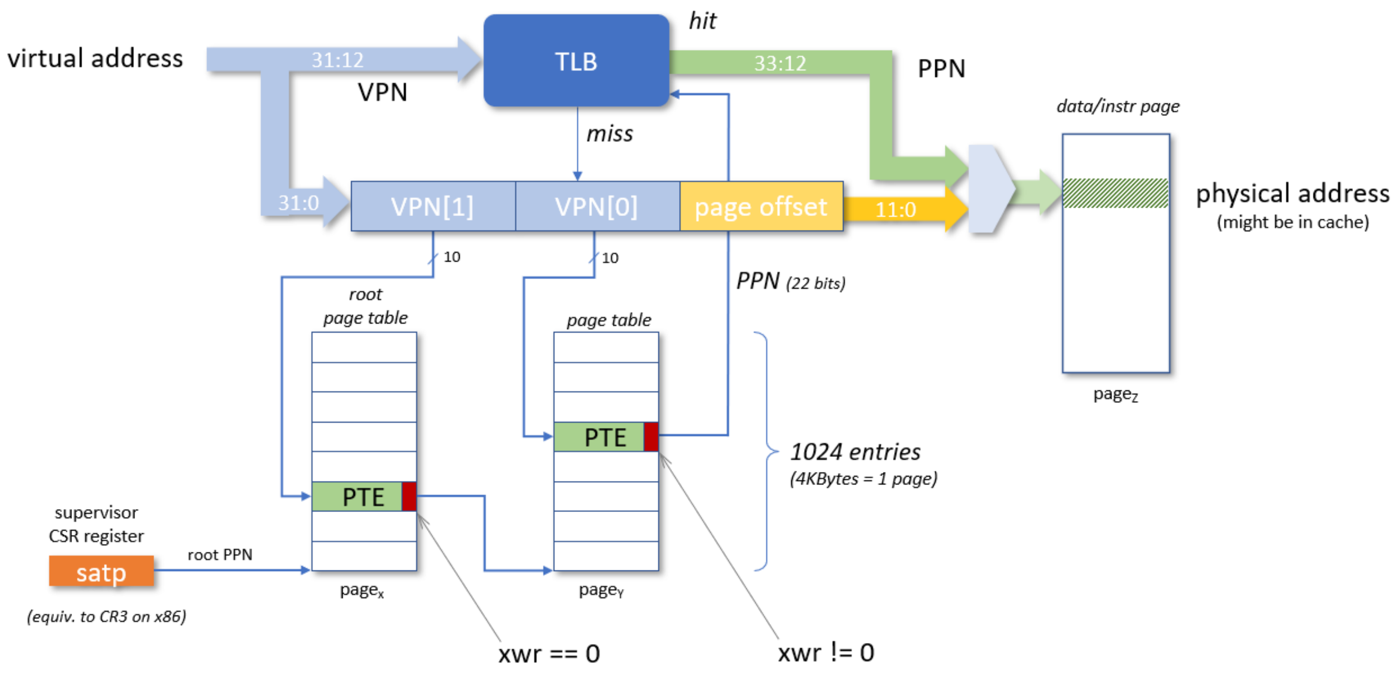
Since Sv32 is for a 32-bit system, the virtual address is 32 bits. The pages are 4Kbytes, and so the lower 12 bits of the address are an offset into a page. The next 20 bits are the virtual page number, but these are divided into two 10-bit values as shown. The physical address format is bigger, with the same offset bits for a 4Kbyte page, but 22 bits for the physical page number. For a 32-bit architecture, only 4GBytes of virtual space can be accessed by an individual process. However, since there are multiple processes, these can be mapped to a large physical memory space. In this case, to a 16Gbyte space. Thus, a system can choose to add up to 16GBytes of memory, even the architecture can only address 4Gbytes.

The page table entry (PTE) consists, firstly, of the 22 bits of a physical page number. (Note that bits 33 down to 12 of the physical address are mapped to bits 31 down to 10 of the PTE.) A couple of bits are reserved, and then some control and status bits are defined for privilege and permission control, some TLB cache status and other status bits for use by the system.

Firstly, the *valid* bit just marks the PTE as containing a translation. This is used for both the table entry in RAM and when in the TLB. The next three bits defined the access permissions and hierarchy of the table entry—i.e., whether it points to another table or not. The diagram shows what each combination means, with greyed out entries reserved. The values are mostly combinations of read, write and/or execution permissions. If all three bits are 0, however, then the entry is a pointer to the next table in a table hierarchy, and the PPN is the physical page number where that table is located.

The user bit indicates that the page has permissions for access in user privilege mode (the lowest privilege for RISC-V). The global bit is used if the system supports multiple address spaces, and the page is mapped to all the spaces. The accessed bits, along with the dirty bit is used, much as for the caches described in the previous article. The accessed bit used for choosing which pages to swap out when a new page requires to be loaded. This could be from the TLB to RAM, or from RAM to the swap space on disk. Similarly, the dirty bit indicates that the page has been written since last written back to the layer below, thus indicating whether it needs to be written back not if swapped out.

Having defined the format of the various addresses and page table entries, let's look at how this all works together. The diagram below shows how an Sv32 two stage lookup might work.



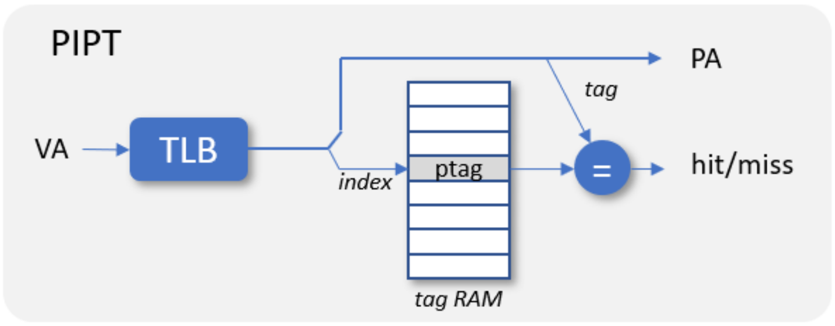
When the processor does a load or store, the VPN bits from the virtual address are presented to the TLB. If the translation is in the TLB (a hit), then the physical page number (PPN) is returned. The physical address to memory is then the PPN concatenated with the 12 bits of offset. The data at this physical address might be in the data (or instruction) cache, or the cache system may have to retrieve this from main memory, just as described in the previous article.

If the entry is not in the TLB (a miss), then a table walk must be started to look up the translation in RAM. In a RISC-V system the root table's page number for the running process is stored in a supervisor CSR register called *satp* (Supervisor Address Translation Protection). The equivalent in an x86 system is the CR3 register. This *satp* register will be updated at each context swap to point to the tables for the process being restarted. The top bits of the virtual address are used to index into the first table. The entry there will have the *xwr* bits set to 0 (unless it is a 'super page'), and the PPN in the entry points to the next table's physical page where the actual translation resides. The lower VPN bits then index into that page to get the PPN for the access. This would be loaded into the TLB, possibly unloading a resident entry first, and the process continue as if for a hit. It is at this point that the access permissions are checked, and an exception/trap/interrupt raised, if the checks fail, so that the operating system can intervene as appropriate. Needless to say, the access is not completed in this circumstance.

So that's how, in general, an MMU works, using real-world formats, and there's nothing more to discuss, right? Well, there is one more thing where we need to go back to the cache and work out how the virtual memory, the MMU and the data/ instruction caches interact.

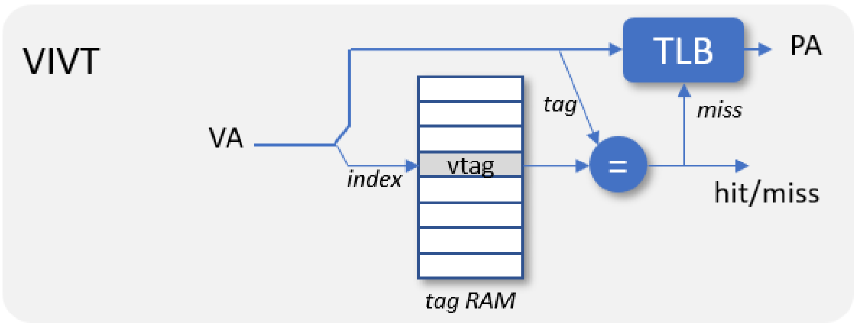
Virtual Memory and Caches

Until now we have looked at virtual memory independently of a memory sub-system's caches. When, in the previous article, we looked at caches it was assumed that all addresses were physical addresses. Looking at both together, the most obvious thing to do to combine the two would be to first translate virtual to physical addresses and then send the physical address accesses through the cache. This is a valid arrangement, and is known as a 'physical index, physical tag' architecture (PIPT). The diagram below shows this arrangement.



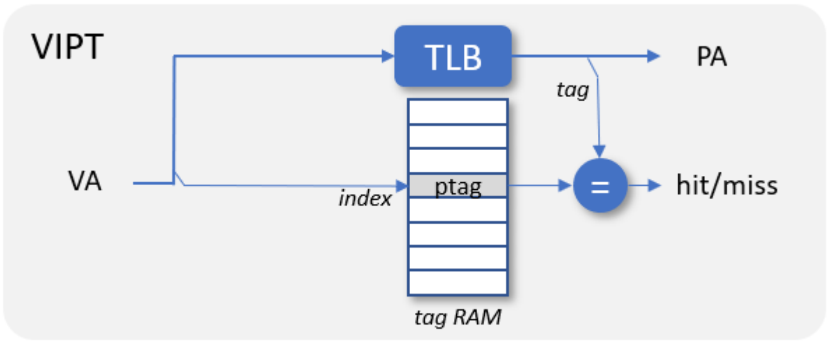
The disadvantage of this arrangement is that (assuming the VA hits in the TLB) the TLB must be looked up first, and only then the cache inspected for a hit on the physical tag. The two lookup latencies are added.

A solution might be to work on the virtual addresses instead. This arrangement is known as 'virtual index, virtual tag' or VIVT. The diagram below shows this architecture.



This arrangement avoids the TLB latency if the data is in the cache, as the hit cache entry is a virtual address and physical memory references are not needed. However, it has a couple of problems. Firstly, the TLB entry must still be looked up for the checking the permission bits. Secondly the cache would need flushing each time there was a context switch as processes can use the same virtual addresses and there would be a clash if a matching address from a different process remained in the cache. PIDs/context numbers as part of the tag might alleviate that somewhat, but that is more complicated and, whilst a give process is running, the cache entries for a different process will definitely not be hit, wasting cache space for the running process.

A compromise solution is where the virtual address is used for the cache index, but the physical address is used for the tag. This is known as 'virtual index, physical tag' (VIPT). The diagram below shows this arrangement.



In this arrangement both the cache and TLB lookups are done in parallel. Since the physical address's tag was stored in the cache tag RAM, then the physical address returned from the TLB can be compared with the physical address from the cache tag lookup and a hit/miss flagged. This architecture then reduces the latency to just that of the longest between the TLB and cache lookups which, hopefully, are comparable.

Conclusions

In this article we have looked at virtual memory and what problems it is trying to solve for a multi-tasking environment. A virtual view of memory is given to each task/process/program so that the software does not need to be aware of the resource clashes caused by running lots of different code concurrently.

Virtual memory support is implemented by dividing the virtual and physical address spaces into pages, and then the virtual pages are mapped to physical pages, dynamically. Page tables are resident in memory to keep track of these mappings, with page table sets for each running process. The logic to do the table lookup and virtual to physical address translations are implemented in an MMU, which also checks for access permissions. The MMU might keep a cache of translations in a TLB in order to speed up the process, just as for a data or instruction cache that speeds up main memory accesses.

We looked at a real-world example of virtual memory formats for addresses and page tables to consolidate the theory, using the RISC-V specification. We also had to discuss how the data/instruction caches interact with the virtual memory functionality in order to get an efficient system.

Over these two articles I've summarised the main aspects of a typical memory sub-system and it might seem that every time a feature was added to solve a particular problem a new one was added which need solving. For example, coherency problems caused by caches in a distributed multi-processor environment, or the positioning of VM translations with respect to those caches to avoid unnecessary latencies. A processor-based environment, as we saw historically, would function without these systems, but not very efficiently. It is often the case that engineering is about optimising systems with added complexity to get the most from the capabilities. Sometimes a solution causes a new issue which must be solved but, over time, a solution is found to all of these to get a system with best performance.

If you never need to work directly on, or with, memory sub-systems I hope this and the previous article will act as a case-study on how systems need to be refined in engineering to get a better solution, even if that adds complexity. If you work within a processor-based systems (as is likely) an awareness of the memory sub-system functionality is useful when designing within that system (in either logic or software realms) and I hope these summaries are useful to that end. I have, in the past, worked more directly on designs that implement MMU functionality but, even in completely different environments where the design space was outside of the processor systems and sub-systems, I have had to design DMA functionality that can traverse over buffers in contiguous virtual memory, but distributed in physical memory, and solve issues caused by cache coherency in order to make it function correctly. This would not have been possible without a knowledge of the functionality discussed in these articles.