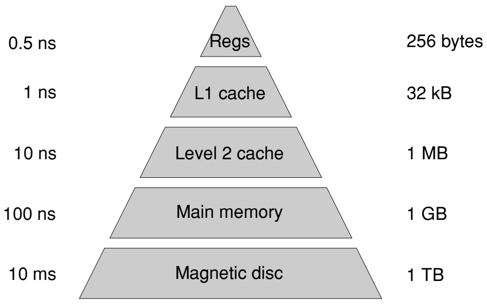
ARMs and architecture - Pt. 3

Nicholas Cutler

The first two parts of this series have concentrated on the processor, looking firstly at the design of instruction sets, and then at the internal optimisations to make them run faster. The remaining component in computer architecture is the memory system, and in most cases it is at least as important as the processor. Very often, improvements in memory have as much effect on real programs as the processor alone.

This is an intuitive concept as any practical computer needs a reliable memory of a decent size. This is provided by semiconductor memories, which enabled the micro-computer revolution of the 1980s. However, performance improvements in memory have not kept up with the processor. In an ideal world, users and programmers alike would want a large memory in which any word would be available instantly. This, of course, cannot be achieved in practice, and it most certainly wouldn't come at an affordable price.

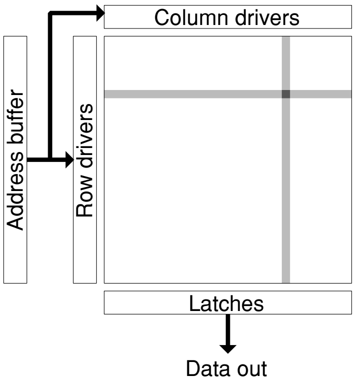
As computer memory can usually only offer two attributes out of large, fast and cheap, the reality is a compromise. To offer the best possible combination, modern computers have a hierarchy of different memory types: the processor's registers hold the variables for the current calculation; the cache holds recently used instructions and data; while the slower main memory can hold the entire program and data set. A typical memory hierarchy is shown in the diagram below. Notice that as we move down a level, the capacity increases, but so does the access time:



There are two main types of semiconductor memory: Static and Dynamic RAM. A single bit of Static RAM is constructed from six transistors, whereas a bit of Dynamic RAM comprises a single transistor and capacitor. The former is faster, occupies more space and consumes more power while active. For this reason it is best suited to cache memory. Alternatively, DRAM is compact and inexpensive, but it is relatively slow. For these reasons it is used in the main memory of nearly all computers. At the bottom of the hierarchy is the magnetic disc. Although the slowest form of memory by a significant margin, it is also the largest and cheapest. Furthermore, it has the advantage of being non -volatile and is therefore suitable for long term storage of data between sessions. There have been various attempts to fill the gap between DRAM and magnetic discs, although improvements in the cost of DRAM and the speed of hard discs have rendered most of these obsolete. The closest contender is flash memory which is increasingly taking over from hard discs.

As mentioned above, nearly all computers use DRAM for the main memory, and memories of a gigabyte or more are increasingly common. The obvious way of building a large memory from DRAM is to have individual cells arranged in a grid with two sets of lines to select a single bit by row and column for reading or writing. The layout of a typical DRAM memory is shown below:

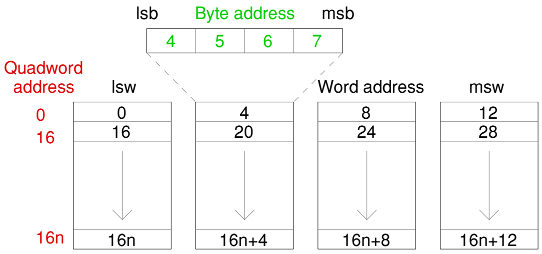
A typical DRAM memory chip might have a capacity of one gigabit, arranged as four



arrays of 256 million cells each. Although this may be further subdivided internally. Such a device can be described as 256Mx4, and four bits can be read or written at once. As the ARM has a 32 bit wide data bus, eight such memory chips would be sufficient to provide 1Gb of RAM. Although reads and writes typically affect a word of 32 bits at a time, memory addresses are usually in bytes, because it is sometimes necessary to use single bytes, for example when working with text strings. The address of a full word must be a multiple of 4 bytes. If only a single byte is required, the whole word is transferred and everything except the desired byte is discarded.

When reading or writing memory, the address is sent in two parts, firstly to select the row, and secondly to select the column. The time to access memory is therefore taken up with the time to send the row and column address, plus the data transfer time. The row access time is about 50ns, but the column access time only about 2.5ns. This difference means that DRAM memory often offers a “fast page mode”, which allows two memory accesses in the same row to happen more quickly. The load and store multiple words instructions on the ARM are designed to exploit this and allow several consecutive words to be transferred without the cost of a full memory access for each. It is also possible to exploit this property when transferring a block from main memory into the cache.

Another technique to speed up memory access is to increase the ammount of data that can be transferred on a single access. Instead of arranging memory into a single bank of 32-bit words, it is possible to have several banks in parallel. On each memory access, every bank will transfer a word each. With a total of four banks, a quadruple word or 128 bits can be transferred at any one time. The memory bandwidth has increased fourfold, even though the speed of the physical RAM has remained the same. An example with the memory arranged into 4 banks of 32 bit words is shown below, demonstrating how such a memory can be addressed as bytes, words, or whole quadwords:



Of course, this is not always beneficial, especially given that each 32-bit instruction word will take just as long to read as a full quadruple word. However, when combined with the cache, whole blocks of data can be transferred more quickly, while the processor continues to read data one word at a time as required. Additionally, some processors are able to work with 64-bit data including version eight of the ARM architecture which has a 64-bit mode (AArch64). Furthermore various extensions allow a processor to take advantage of wide data paths by executing a single instruction on several short data items at once, such as ARM’s Neon, or Intel’s

MMX and SSE. In the case of Neon, a single quadruple word register can be considered as four 32-bit words, eight 16-bit half words or 16 bytes. An instruction such as an addition can be carried out on each of the items in parallel.

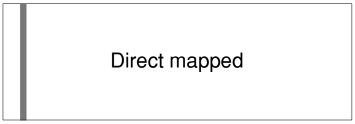
While such techniques for increasing memory bandwidth are helpful, they do not remove the need for a cache to act as a buffer between the processor and memory. Indeed they often complement the cache by allowing data to be fetched efficiently in blocks, even though the processor only needs one word at a time. The previous part of this series has already introduced the concept, but this part will now go on to examine the details:

Recall that the purpose of the cache is to provide fast access to recently used data and instructions. Also remember that the ARM 3 had just 4kb of cache memory, while even now the upper limit is about 64 kb. Clearly this will only store the most trivial of programs, so frequent replacement of the contents will be necessary. The work of transferring data between memory and the cache happens automatically and is invisible to even the programmer. However, some knowledge of the characteristics of the memory system makes it possible to optimise programs.

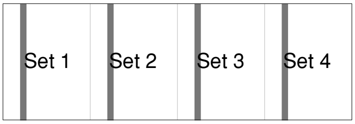
Of course, transferring and replacing data at the level of individual words is too complicated, while it also fails to take advantage of the fast page mode offered by DRAM. For this reason most caches deal in blocks, where a typical size is 16 words, or 64 bytes. When the processor requests a specific word, the entire block containing the requested word is fetched (the block address is just the byte or word address with the bottom six bits set to zero). In most cases this is a sensible approach to take as memory accesses, especially for instructions, are often sequential. Although early processors sometimes shared the cache between instructions and data, it is now more usual to have separate caches for each, to take advantage of the more predictable nature of instruction fetches.

Having determined that a block must be fetched from memory, the next question is where it should go within the cache. It is entirely possible that the hardware could look up in a table to determine, say, the oldest block in the cache and replace it with the incoming data. Such a cache is said to be fully associative, and this is the most flexible approach. However, to simplify the cache control, there are often some limits on where the block can be placed. At the other end of the scale is a direct mapped cache where there is only one possible location for the block, determined entirely by its address. Somewhere in between are set associative caches which split the total space up into a number of sets. Each incoming block of data can only be put into a specific location, but within any set. This offers some flexibility while also reducing the amount of hardware necessary to support the cache. Increasing the associativity will increase the chance that the requested data is in the cache. In practice a change to full associativity has a similar impact to doubling the overall size with 2-way associativity. The placement of 64 byte blocks in a small 4 kb cache is shown below. The requested word is &180C8, in block &603 (found by dividing the address the block size.) In a direct mapped cache, this has a unique position which is the remainder when dividing the block number, &603, by 64 (the number of lines in the cache). So in this case the block must go in line 3. For a set associative cache the calculation is similar, although there are fewer lines in each set:

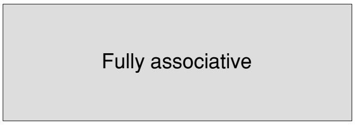
For anything except a direct mapped cache, there is a choice of which block gets discarded



Block &603 is placed in slot 3



Block &603 can go in slot 3 of any set



The block can be placed anywhere

to make way for the new one. One plausible strategy is to replace the oldest one, although it fails to take account of blocks which are in frequent use throughout the execution of the program (perhaps global variables or common subroutines). Another possibility is to discard the block which has been used least recently. While this takes account of usage patterns, it is possible to think of situations where this too fails. In practice, however, this is often the best strategy. The simplest option, replacing blocks randomly, is the worst but the differences in performance on real tasks are surprisingly small.

Having looked at some of the theory, it is useful to consider some real examples. The table below summarises the caches on three ARM processors. In particular look at the miss rate which shows the proportion of loads which are not in the cache. Although these figures are estimates based on typical programs and should be taken with some caution, they show that even relatively small caches have a beneficial effect on performance:

Processor	**Inst. cache**	**Data cache**	**Miss rate**	
			Inst.	**Data**
ARM 3	4 kb unified/64 way		10% overall	
StrongARM	16 kb/16 way	16 kb/16 way	0.4%	4.1%
Cortex-A53	32 kb/2 way	32 kb/4 way	0.14%	3.8%

However, one surprise from these figures is the low degree of associativity on the newest processor. This is partly because the Cortex series adds a second level cache, reducing the impact of a miss. Without this, the ARM 3 and the StrongARM had to interface directly with relatively slow main memory. The high associativity was designed to gain the maximum benefit from the cache.

Where present, such as on the Cortex series, and other processors designed for higher performance, the second level of cache works in a very similar way to the first level. Although slower, it compensates for this with a much greater capacity, up to 1 Mb, and a higher degree of associativity, again because the penalty for a miss is correspondingly greater. The size of a single block may be larger, but the Cortex-A53 keeps the 64 byte blocks as this is the most efficient option for all but the largest caches. While larger blocks improve miss rates up to a point, very large blocks simply do not offer enough fine control over the cache content.

So far this explanation of caches has only considered loads: instructions which transfer data from the memory to the processor. The situation with store instructions is slightly more complex because of the need to ensure that two copies of the data (in the cache and main memory) are kept up to date. One simple way of doing this is to write to the cache, and then to write the same data to main memory, ensuring that both copies remain consistent. This is known as a ‘write through cache’. The disadvantage is that the processor has to wait for the data to be saved to main memory before proceeding. If there are many store instructions in sequence then there is little benefit to be gained from the data cache. The other main technique is to add a buffer to hold data until it can be stored in main memory. This works well in the usual case with relatively few stores, allowing the processor to fetch instructions from the cache, while the data is written to memory. This also highlights another advantage of keeping a separate instruction cache: storing data to this space becomes impossible, precluding the use of self -modifying code.

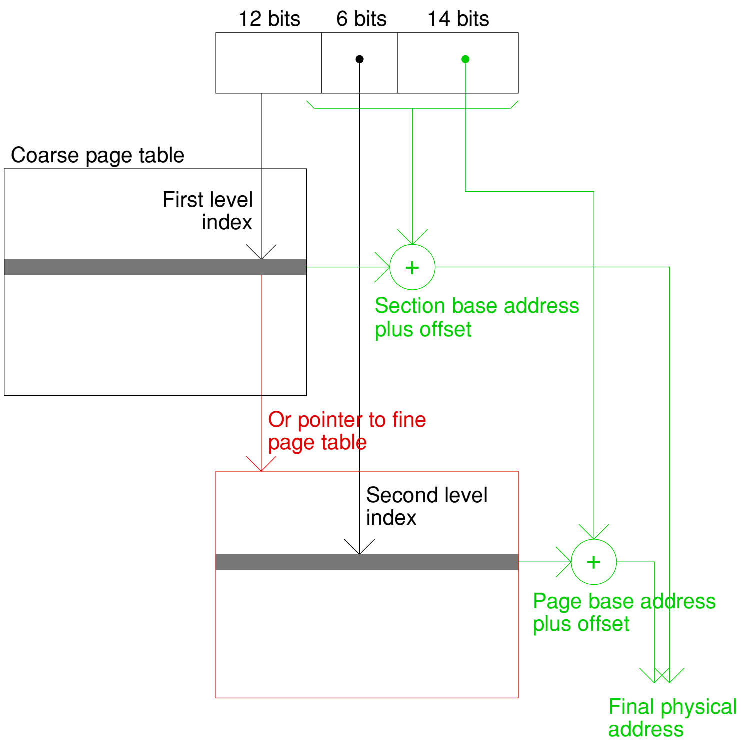
The final component of the memory system is the concept of addressing, providing a way of requesting a specific location, and keeping track of data present in the cache. In the earliest computers there was a direct correspondence between the address and a single memory location. However, as memories grew it became necessary to map between the virtual address used by the processor and the physical location in memory. In the BBC micro, for example, it was often necessary to switch between banks of sideways ROM when changing language or filing system. Each was mapped to the same location in the virtual address space, providing a consistent view of memory to the programmer. Even though there could be several such banks in the machine, only one was active at once. The first Archimedes machines had the opposite problem: a larger address space than the available memory. Here portions of the available RAM could be mapped to different places in the address space, allowing important areas like the video memory or the current task to appear at a consistent location regardless of how much memory might be installed, with large parts of the virtual address space remaining unused.

To map between the virtual and physical address, the memory is split up into a number of pages. The high order bits of the address usually specify a page number, and the low order bits an offset within the page. The page number is used as an index to read the physical address from a table. The final address is given by the base of the physical page plus the offset. On recent RISC OS machines, pages of 16 kb are common, so the page number is given by the top 18 bits of the virtual address, and the offset by the bottom 14 bits.

One advantage of this table driven scheme is that switching between tasks is relatively easy: only the page table needs to be updated, rather than moving large chunks of data around the memory. The disadvantage is that the page table can get quite large (2^{18} entries for a machine with a 32 bit address space in 16 kb pages). Although the address translation is hidden from the application programmer, and performed in hardware by the memory controller, there is still a penalty. To speed this up, recently used page table entries, such as those for the current task, are often cached by the memory controller. This area, known as the ‘translation lookaside buffer’ works much like the processor’s cache.

To further reduce the size of the page tables, a hybrid scheme which is able to handle large pages of a megabyte or more, alongside the more usual small pages, is sometimes used. In such a scheme the high order portion of the address refers to an entry in the page table as before. However, this might be either the address of a large page, or a pointer to a further table of small pages. Such a scheme is supported by the newer ARM processors, although RISC OS has not made use of it hitherto. The operation of address translation with large pages of 1 Mb and small pages of 16 kb is summarised below:

The real advantage of a large virtual address space, is the ability to present a consistent view to the programmer, while data can reside anywhere in the physical memory. Such a scheme allows the most frequently used blocks to be mapped into the cache and



referenced by virtual address. One further advantage of the page based mapping scheme is that it can provide a degree of protection by preventing unauthorised access to parts of the memory. This might include: preventing one task from writing to another's space; ensuring pages containing programs are read only; and ensuring that the user programs cannot access pages used by the operating system. While RISC OS has not made much use of memory protection it remains important for multi-user operating systems.

In summary this article has concluded the series on computer architecture by looking at the memory system. The predomient type of memory is DRAM which is both compact and inexpensive, but relatively slow. Thus, a combination of memories is necessary to offer a good compromise between speed and capacity. At least one level of cache acts as a buffer between the processor and main memory, while the idea of a virtual address space brings the entire memory hierarchy together. Address translation allows the memory to be split up into pages and mapped into the available RAM, and simplifies the process of switching between tasks.

When the internal design of the processor is considered too, a lot of complexities have been hidden. Even so, understanding computers at this level makes it possible to optimise programs. In future, an appreciation of these details will be essential to make the most of newer architectures.

ncc25@srcf.net

It is said that the first instruction set architecture that you learn should be an exemplar, like first love. Nicholas thinks this was once true of the ARM, but less so after recent extensions.