Tales from a lean programmer.

An overview of direct memory access

April 27, 2014 by David 3 Comments

Introduction

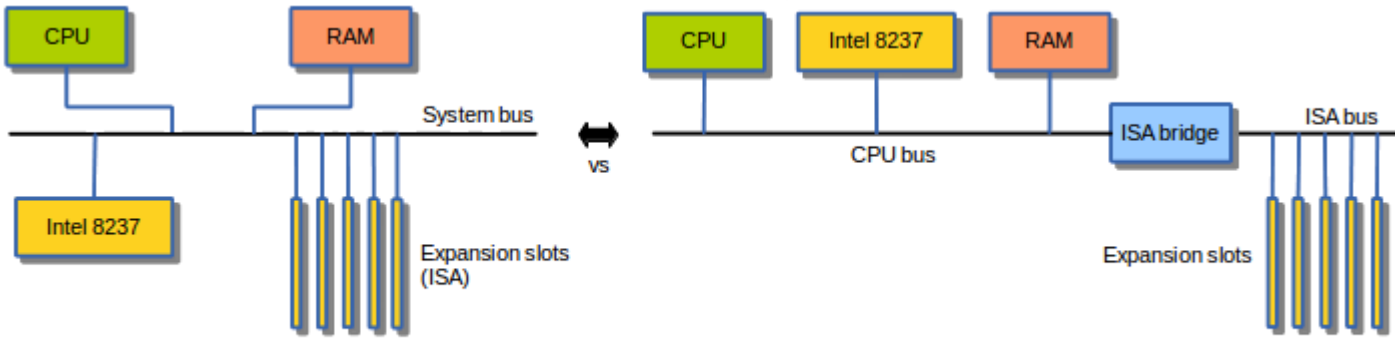
Direct memory access (DMA) is conceptually easy, but without experience in hardware design or driver development it can be cumbersome to understand. In this blog post I will explain what DMA is and how it evolved during the last decades. My goal is to make it comprehensible especially for people without experience in hardware design or driver development.

DMA allows computer devices of certain hardware sub-systems to directly access system memory and other device's memory independently of the CPU. This enables the CPU to keep on working concurrently on other task while long lasting memory operations take place; considerably boosting overall system performance. DMA is used by different hardware like graphics cards, sound cards, network cards and disk drive controllers. DMA is rather a concept than a specific technology. There is no specification which describes in detail how DMA transfers work. Even on the contrary, the concept of directly accessing memory without CPU interaction is employed in many different hardware sub-systems in today's computers. The most typical application is communicating with peripheral devices plugged into a bus system like *ATA*, *SATA*, *PCI* or *PCI Express*. Beyond that, DMA transfers are used for intra-core communication in micro processors and even to copy data from the memory of one computer into the memory of another computer over the network via *remote DMA* (don't mix up this technology with NVIDIA's new *GPUDirect* RDMA feature).

To give a concrete example, imagine you're playing an open world computer game which loads new game assets on demand from your hard disk. Large amounts of game data must be copied over from hard disk into system RAM. Without DMA the CPU would be actively involved in each and every memory transfer operation. Consequently, less computing time would be left for other game play related tasks like AI or physics. In times of multi-core processors this seems less like a problem. However, as data volumes and work load sizes are ever growing, off-loading large memory transfer operations from the CPU is also today absolutely essential in order to achieve high system performance.

How DMA evolved over time

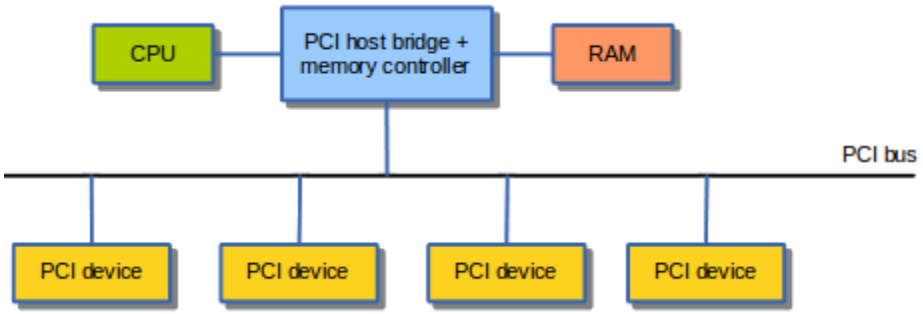
In my experience many software people think that DMA nowadays still works as it did in the old days. I guess this is because it's the more intuitive way to think about DMA. Back then, extension devices did not actively take part in DMA transfers, but there was a *DMA controller* (e.g. the Intel 8237, first used in the IBM PC in 1981) which enabled DMA transfers between system memory and device I/O over the good old *Industrial Standard Architecture* (*ISA*) bus. The DMA controller could be programmed by the CPU to perform a number of memory transfers on behalf of the CPU. This way of accomplishing DMA transfers is also known as *third party DMA*. At that time the *system bus* was identical to the ISA expansion bus. To account for reduced bus performance in situations where CPU and DMA controller needed to access the bus simultaneously, different DMA modes (*cycle stealing*, *transparent* and *burst*) could be used. When the first IBM AT clones came out, the expansion bus got physically separated from the system bus using an *ISA bridge*. This was necessary because the AT clones had CPUs running at higher frequencies than the expansion bus. In the figure below the single bus and the separated bus architectures are depicted.



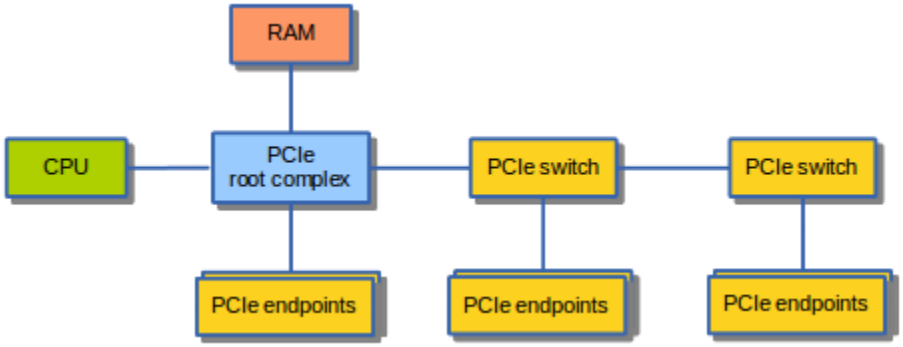
Single system bus architecture

Separated CPU and ISA bus architecture

With the introduction of the conventional *Peripheral Component Interface* (*PCI*) bus architecture in 1992, the DMA controller became obsolete because of a technique called *bus mastering*, or *first party DMA*. PCI DMA transfers were implemented by allowing only one device at a time to access the bus. This device is called the *bus master*. While the bus master holds the bus it can perform memory transfers without CPU interaction. The fundamental difference between bus mastering and the use of a DMA controller is that DMA compatible devices must contain a *DMA engine* driving the memory transfers. As multiple PCI devices can master the bus, an *arbitration* scheme is required to avoid that more than one device drives the bus simultaneously. The advantage of bus mastering is a significant latency reduction because communication with the third party DMA controller is avoided. Additionally, each device's DMA engine can be specifically optimized for the sort of DMA transfers it performs.



Today's computers don't contain DMA controllers anymore. If they do so, it's only to support legacy buses like e.g. ISA, often by simulating an ISA interface using a *Low Pin Count* (*LPC*) bus bridge. In 2004 the PCI successor and latest peripheral computer bus system *PCI Express* (*PCIe*) was introduced. PCIe turned the conventional PCI bus from a true bus architecture, with several devices physically sharing the same bus, into a *serial*, *packet-switched*, *point-to-point*architecture; very similar to how packet-switched networks function. PCIe connects each device with a dedicated, bi-directional link to a PCIe switch. As a result, PCIe supports *full duplex* DMA transfers of multiple devices at the same time. All arbitration logic is replaced by the packet routing logic implemented in the PCIe switches. While PCIe is entirely different to PCI on the hardware level, PCIe preserves backwards compatibility with PCI on the driver level. Newer PCIe devices can be detected and used by PCI drivers without explicit support for the PCIe standard. Though, the new PCIe features cannot be used of course.



DMA from a driver developer's perspective

Now you know what DMA is and how it fits into a computer's hardware architecture. So let's see how DMA can be used in practice to speed up data heavy tasks. Since the dawn of DMA the driver (software) must prepare any peripheral DMA transfers, because only the operating system (OS) has full control over the memory system (we see later why this is important), the file system and the user-space processes. In the first step, the driver determines the source and destination memory addresses for the transfer. Next, the driver programs the hardware to perform the DMA transfer. The major difference between PCI/PCIe DMA and legacy ISA DMA is the way a DMA transfer is initiated. For PCI/PCIe no uniform, device independent way to initiate DMA transfers exists anymore, because each device contains its own, proprietary DMA engine. In contrast, the legacy DMA controller is always the same.
First, the peripheral device's DMA engine is programmed with the source and destination addresses of the memory ranges to copy. Second, the device is signaled to begin the DMA transfer. Fair enough, but how can the driver know when the DMA transfer has finished? Usually, the device raises *interrupts* to inform the CPU about transfers that have finished. For each interrupt an *interrupt handler*, previously installed by the driver, is called and the finished transfer can be acknowledged accordingly by the OS (e.g. signaling the block I/O layer that a block has been read from disk and control can be handed back to the user-space process which requested this block). Back in the times of high latency spinning disks an slow network interfaces this was sufficient. Today, however, we've got solid state disks (SSD) and gigabit, low-latency network interfaces. To avoid completely maxing out the system by a vast number of interrupts, a common technique is to hold back and queue up multiple interrupts on the device until e.g. a timeout triggers, a certain number of interrupts are pending or any other condition suiting the application is met. This technique is known as *interrupt coalescing*. Obviously, the condition is always a trade-off between low latency and high throughput. The more frequently new interrupts are raised, the quicker the OS and its waiting processes are informed about finished memory transfers. However, if the OS is interrupted less often it can spend more time on other jobs.
DMA seems to be a nice feature in theory, but how does transferring large continuous memory regions play together with *virtual memory*? Virtual memory is usually organized in chunks of 4 KiB, called *pages*. Virtual memory is continuous as seen from a process' point-of-view thanks to *page tables* and the *memory management unit* (*MMU*). However, it's non-continuous as seen from the device point-of-view, because there is no MMU between the PCIe bus and the memory controller (well, some CPUs have an *IO-MMU* but let's keep things simple). Hence, in a single DMA transfer only one page could be copied at a time. To overcome this limitation OS usually provide a *scatter/gather* API. Such an API chains together multiple page-sized memory transfers by creating a list of addresses of pages to be transferred.

Take home message

DMA is an indispensable technique for memory-heavy, high-performance computing. Over the last decades, the entire bus system and DMA controller concept was superseded by moving the DMA controller into the devices and using a point-to-point bus architecture. This reduced latency, made concurrent DMA transfers possible and allowed for device specific DMA engine optimizations. For the drivers less has changed. They are still responsible for initiating the DMA transfers. Though, today, instead of programming a DMA controller in a device independent way, drivers must program device specific DMA engines. Therefore, programming DMA transfers and processing DMA status information can look very different depending on the device.