WIKIPEDIA

Memory-mapped I/O

Memory-mapped I/O (**MMIO**) and **port-mapped I/O** (**PMIO**) (which is also called ***isolated I/O***) are two complementary methods of performing input/output (I/O) between the central processing unit (CPU) and peripheral devices in a computer. An alternative approach is using dedicated I/O processors, commonly known as channels on mainframe computers, which execute their own instructions.

Memory-mapped I/O uses the same address space to address both memory and I/O devices. The memory and registers of the I/O devices are mapped to (associated with) address values. So when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the data bus to the desired device's hardware register. To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and must not be available for normal physical memory. The reservation may be permanent or temporary. An example of the latter is found in the Commodore 64, which uses a form of memory mapping to cause RAM or I/O hardware to appear in the `$D000`-`$DFFF` range.

Port-mapped I/O often uses a special class of CPU instructions designed specifically for performing I/O, such as the `in` and `out` instructions found on microprocessors based on the x86 and x86-64 architectures. Different forms of these two instructions can copy one, two or four bytes (`outb`, `outw` and `outl`, respectively) between the EAX register or one of that register's subdivisions on the CPU and a specified I/O port which is assigned to an I/O device. I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, or an entire bus dedicated to I/O. Because the address space for I/O is isolated from that for main memory, this is sometimes referred to as isolated I/O.

Contents

Overview

Different CPU-to-device communication methods, such as memory mapping, do not affect the direct memory access (DMA) for a device, because, by definition, DMA is a memory-to-device communication method that bypasses the CPU.

Hardware interrupts are another communication method between the CPU and peripheral devices, however, for a number of reasons, interrupts are always treated separately. An interrupt is device-initiated, as opposed to the methods mentioned above, which are CPU-initiated. It is also unidirectional, as information flows only from device to CPU. Lastly, each interrupt line carries only one bit of information with a fixed meaning, namely "an event that requires attention has occurred in a device on this interrupt line".

I/O operations can slow memory access if the address and data buses are shared. This is because the peripheral device is usually much slower than main memory. In some architectures, port-mapped I/O operates via a dedicated I/O bus, alleviating the problem.

One merit of memory-mapped I/O is that, by discarding the extra complexity that port I/O brings, a CPU requires less internal logic and is thus cheaper, faster, easier to build, consumes less power and can be physically smaller; this follows the basic tenets of reduced instruction set computing, and is also advantageous in embedded systems. The other advantage is that, because regular memory instructions are used to address devices, all of the CPU's addressing modes are available for the I/O as well as the memory, and instructions that perform an ALU operation directly on a memory operand (loading an operand from a memory location, storing the result to a memory location, or both) can be used with I/O device registers as well. In contrast, port-mapped I/O instructions are often very limited, often providing only for simple load-and-store operations between CPU registers and I/O ports, so that, for example, to add a constant to a port-mapped device register would require three instructions: read the port to a CPU register, add the constant to the CPU register, and write the result back to the port.

As 16-bit processors have become obsolete and replaced with 32-bit and 64-bit in general use, reserving ranges of memory address space for I/O is less of a problem, as the memory address space of the processor is usually much larger than the required space for all memory and I/O devices in a system. Therefore, it has become more frequently practical to take advantage of the benefits of memory-mapped I/O. However, even with address space being no longer a major concern, neither I/O mapping method is universally superior to the other, and there will be cases where using port-mapped I/O is still preferable.

Memory-mapped I/O is preferred in x86-based architectures because the instructions that perform port-based I/O are limited to one register: EAX, AX, and AL are the only registers that data can be moved into or out of, and either a byte-sized immediate value in the instruction or a value in register DX determines which port is the source or destination port of the transfer.[1][2] Since any general-purpose register can send or receive data to or from memory and memory-mapped I/O, memory-mapped I/O uses fewer instructions and can run faster than port I/O. AMD did not extend the port I/O instructions when defining the x86-64 architecture to support 64-bit ports, so 64-bit transfers cannot be performed using port I/O.[3]

Memory barriers

Since the caches intermediate accesses to memory addresses, data written to different addresses may reach the peripherals' memory or registers out of the program order, i.e. if software writes data to an address and then writes data to another address, the cache write buffer does not guarantee that the data will reach the peripherals in that order.[4] It is the responsibility of the software to include memory barrier instructions after the first write, to ensure that the cache buffer is drained before the second write is executed.

Memory-mapped I/O is the cause of memory barriers in older generations of computers, which are unrelated to memory barrier instructions. The 640 KB barrier is due to the IBM PC placing the Upper Memory Area in the 640–1024 KB range within its 20-bit memory addressing. The 3 GB barrier and PCI hole are manifestations of this with 32-bit memory addressing; with 64-bit memory addressing these are usually no longer problems on newer architectures.

Examples

A sample system memory map

Address range (hexadecimal)	Size	Device
0000–7FFF	32 KiB	RAM
8000–80FF	256 bytes	General-purpose I/O
9000–90FF	256 bytes	Sound controller
A000–A7FF	2 KiB	Video controller/text-mapped display RAM
C000–FFFF	16 KiB	ROM

A simple system built around an 8-bit microprocessor might provide 16-bit address lines, allowing it to address up to 64 kibibytes (KiB) of memory. On such a system, the first 32 KiB of address space may be allotted to random access memory (RAM), another 16 KiB to read only memory (ROM) and the remainder to a variety of other devices such as timers, counters, video display chips, sound generating devices, etc.

The hardware of the system is arranged so that devices on the address bus will only respond to particular addresses which are intended for them, while all other addresses are ignored. This is the job of the address decoding circuitry, and that establishes the memory map of the system. As a result, system's memory map may look like in the table on the right. This memory map contains gaps, which is also quite common in actual system architectures.

Assuming the fourth register of the video controller sets the background colour of the screen, the CPU can set this colour by writing a value to the memory location A003 using its standard memory write instruction. Using the same method, graphs can be displayed on a screen by writing character values into a special area of RAM within the video controller. Prior to cheap RAM that enabled bit-mapped displays, this character cell method was a popular technique for computer video displays (see Text user interface).

Basic types of address decoding

Address decoding types, in which a device may decode addresses completely or incompletely, include the following:

Complete (exhaustive) decoding

1:1 mapping of unique addresses to one hardware register (physical memory location). Involves checking every line of the address bus.

Incomplete (partial) decoding

n:1 mapping of n unique addresses to one hardware register. Partial decoding allows a memory location to have more than one address, allowing the programmer to reference a memory location using n different addresses. It may also be done to simplify the decoding hardware by using simpler and often cheaper logic that examines only some address lines, when not all of the CPU's address space is needed. Commonly, the decoding itself is programmable, so the system can reconfigure its own memory map as required, though this is a newer development and generally in conflict with the intent of being cheaper. Synonyms: foldback, multiply mapped, partially mapped, address aliasing.[5][6]

Linear decoding

Address lines are used directly without any decoding logic. This is done with devices such as RAMs and ROMs that have a sequence of address inputs, and with peripheral chips that have a similar sequence of inputs for addressing a bank of registers. Linear addressing is rarely used alone (only when there are few devices on the bus, as using purely linear addressing for more than one device usually wastes a lot of address space) but instead is combined with one of the other methods to select a device or group of devices within which the linear addressing selects a single register or memory location.

Port I/O via device drivers

In Windows-based computers, memory can also be accessed via specific drivers such as DOLLx8KD which gives I/O access in 8-, 16- and 32-bit on most Windows platforms starting from Windows 95 up to Windows 7. Installing I/O port drivers will ensure memory access by activating the drivers with simple DLL calls allowing port I/O and when not needed, the driver can be closed to prevent unauthorized access to the I/O ports.

Linux provides the `pcimem` utility to allow reading from and writing to MMIO addresses. The Linux kernel also allows tracing MMIO access from kernel modules (drivers) using the kernel's *mmiotrace* debug facility. To enable this, the Linux kernel should be compiled with the corresponding option enabled. *mmiotrace* is used for debugging closed-source device drivers.

See also

- mmap, not to be confused with memory-mapped I/O
- Early examples of computers with port-mapped I/O
  - PDP-8
  - Nova
- PDP-11, an early example of a computer architecture using memory-mapped I/O

    - Unibus, a dedicated I/O bus used by the PDP-11
- Input/output base address
- Bank switching
- Ralf Brown's Interrupt List

References

1. "Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 2A: Instruction Set Reference, A-M" (http://www.intel.com/Assets/PDF/manual/253666.pdf) (PDF). *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. June 2010. pp. 3–520. Retrieved 2010-08-21.
2. "Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 2B: Instruction Set Reference, N-Z" (http://www.intel.com/Assets/PDF/manual/253667.pdf) (PDF). *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. June 2010. pp. 4–22. Retrieved 2010-08-21.
3. "AMD64 Architecture Programmer's Manual: Volume 3: General-Purpose and System Instructions" (http://support.amd.com/us/Processor_TechDocs/24594_APM_v3.pdf) (PDF). *AMD64 Architecture Programmer's Manual*. Advanced Micro Devices. November 2009. pp. 117, 181. Retrieved 2010-08-21.
4. *ARM Cortex-A Series Programmer's Guide. Literature number ARM DEN0013D*. pp. 10–3.
5. Microsoft (December 4, 2001). "Partial Address Decoding and I/O Space in Windows Operating Systems" (http://www.microsoft.com/whdc/system/sysinternals/partialaddress.mspx#EFD).
6. HP. "Address aliasing" (http://docs.hp.com/en/A3725-96022/ch03s03.html).

Retrieved from "https://en.wikipedia.org/w/index.php?title=Memory-mapped_I/O&oldid=805085824"