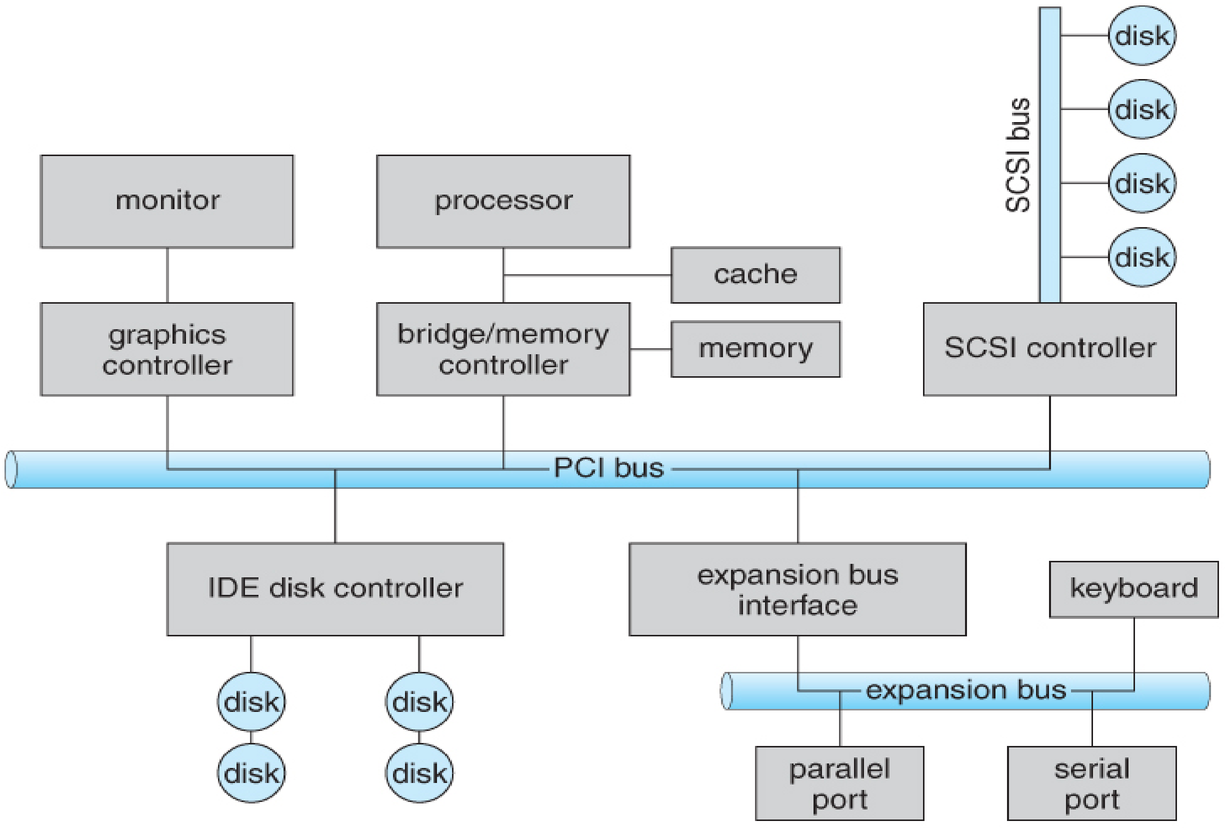
I/O Hardware Control

I/O devices can be roughly categorized as storage, communications, user-interface, and other

Devices communicate with the computer via signals sent over wires or through the air.

- A common set of wires connecting multiple devices is termed a ***bus.*** Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
  - *The **PCI bus** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.)*
  - *The **expansion bus** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)*
  - *The **SCSI bus** connects a number of SCSI devices to a common SCSI controller.*
  - *A **daisy-chain bus,** (not shown) is when a string of devices is connected to each other like beads on a chain and only one of the devices is connected to the host.*

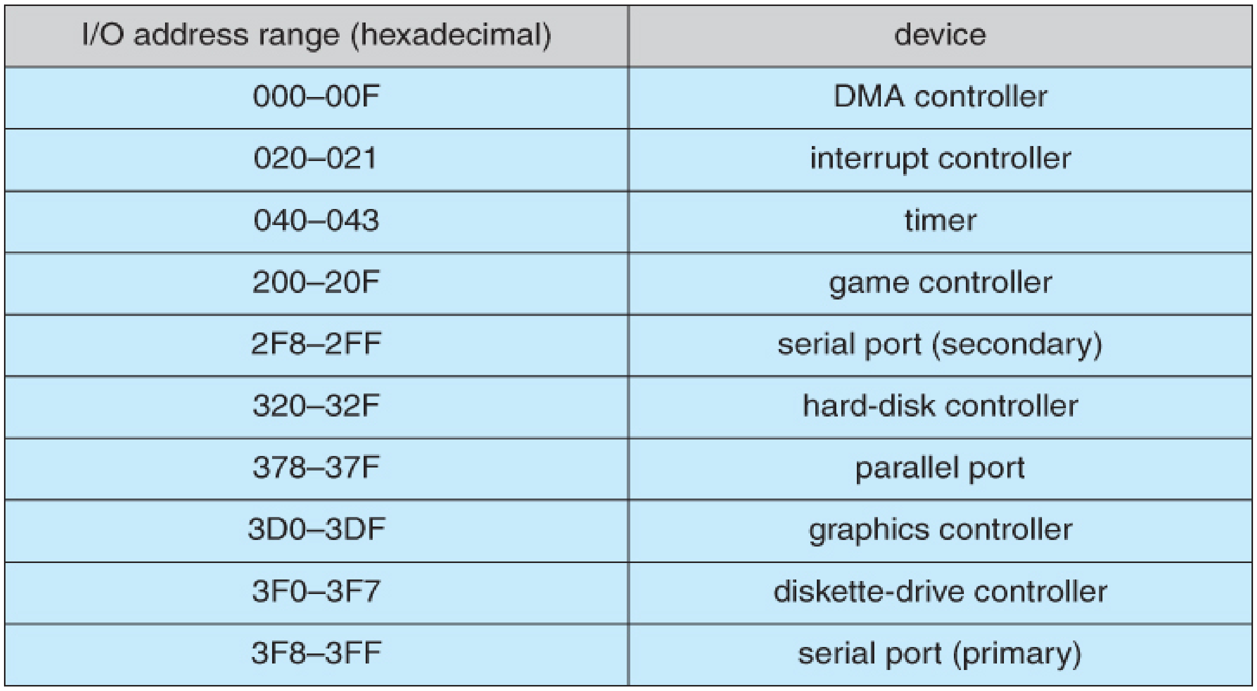


Registers

- One way of communicating with devices is through ***registers*** associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:

- The ***data-in register*** is read by the host to get input from the device.
- The ***data-out register*** is written by the host to send output.
- The ***status register*** has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.
- The ***control register*** has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation.

Some of the most common I/O port address ranges:



I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Polling

- One simple means of device ***handshaking*** involves polling:
  - The host repeatedly checks the ***busy bit*** on the device until it becomes clear.
  - The host writes a byte of data into the data-out register, and sets the ***write bit*** in the command register (in either order.)

- ○ The host sets the ***command ready bit*** in the command register to notify the device of the pending command.
  - ○ When the device controller sees the command-ready bit set, it first sets the busy bit.
  - ○ Then the device controller reads the command register, sees the write bit set, reads the byte of data from the data-out register, and outputs the byte of data.
  - ○ The device controller then clears the ***error bit*** in the status register, the command-ready bit, and finally clears the busy bit, signaling the completion of the operation.

- Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer. It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there

Memory-mapped I/O

- In this case a certain portion of the processor's address space is mapped to the device, and communications occur by reading and writing directly to/from those memory areas.
- Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.
- Memory-mapped I/O can be used either instead of or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.
- A potential problem exists with memory-mapped I/O, if a process is allowed to write directly to the address space used by a memory-mapped I/O device

How data is accessed in memory mapped I/O?

- Memory-mapped I/O uses the same *mechanism* as memory to communicate with the processor, but not the system's RAM. The idea behind memory mapping is that a device will be connected to the system's address bus and uses a circuit called an *address decoder* to watch for reads or writes to its assigned addresses responds accordingly.

  Memory mapped I/O Control

  The logic circuit that contains these registers is called the *device controller*, and the software that communicates with the controller is called a *device driver*.

  +—————-+ +———+

```
                              |  Device controller |            |         |
 +——-+          |              | <———>|  Device  |
 |      | ———->| Control register |            |          |
|  CPU  | <———-|  Status register   |            |       |
 |       | <———>|  Data register     |            |      |
 +——-+          |                    |            |       |
                              +——————-+          +———+
```

- Simple devices such as keyboards and mouse may be represented by only a few registers, while more complex ones such as disk drives and graphics adapters may have dozens.
- Each of the I/O registers, like memory, must have an address so that the CPU can read or write specific registers.
- Some CPUs have a separate address space for I/O devices. This requires separate instructions to perform I/O operations.
- Other architectures, like the MIPS, use *memory-mapped I/O*. When using memory-mapped I/O, the same address space is shared by memory and I/O devices. Some addresses represent memory cells, while others represent registers in I/O devices. No separate I/O instructions are needed in a CPU that uses memory-mapped I/O. Instead, we can perform I/O operations using any instruction that can reference memory.

```
                    +—————+
                    | Address space |
                    |    +——-+ |
                    |  |  ROM  |  |
                    |    +——-+ |
+——-+address|  |        |  |
 |      | ——>|  |   RAM  | |
|  CPU  |       |    |      | |
 |       | <—>|   +——-+ |
+——-+ data |    |       | |
                    |  |  I/O  |  |
                    |    +——-+ |
                    +—————+
```

- On the MIPS, we would access ROM, RAM, and I/O devices using load and store instructions. Which type of device we access depends only on the address used!

```
lw      $ t0, 0x00000004  # Read ROM
sw      $ t0, 0x00000004  # Write ROM (bus error!)

lbu     $ t0, 0x0000ffc1  # Read RAM
sb      $ t0, 0x0000ffc1  # Write RAM

lbu     $ t0, 0xffff0000  # Read an I/O device
sb      $ t0, 0xffff0004  # Write to an I/O device
```

- The 32-bit MIPS architecture has a 32-bit address, and hence an address space of 4 gigabytes. Addresses 0x00000000 through 0xfffeffff are used for memory, and addresses 0xffff0000 - 0xffffffff (the last 64 kilobytes) are reserved for I/O device registers. This is a very small fraction of the total address space, and yet far more space than is needed for I/O devices on any one computer.

- Each register within an I/O controller must be assigned a unique address within the address space. This address may be fixed for certain devices, and auto-assigned for others. (PC plug-and-play devices have auto-assigned I/O addresses, which are determined during boot-up.)

Communicating with a Keyboard Controller

- The keyboard controller consists of two 32-bit registers, of which only a few bits are used.

- The *receiver data register* resides at the fixed memory address 0xffff0004. The low 8 bits of this register contain the ASCII/ISO code of the last key that was pressed.

```
 31            8 7 0
+------------------+
|    unused   | key |
+------------------+
```

  This register is read-only, and can be accessed with a load instruction:

```
# Input ASCII/ISO code of last key to low byte of $ t0
# and clear remaining bits of $ t0
lbu    $ t0, 0xffff0004
```

- As always, we should use named constants rather than hard-code numbers like 0xffff0004.

```
CONS_RECEIVER_DATA          = 0xffff0004

# Input ASCII/ISO code of last key to low byte of $ t0
# and clear remaining bits of $ t0
lbu    $ t0, CONS_RECEIVER_DATA
```

- The lbu (load byte unsigned) instruction should be used instead of lw, since lbu ensures that the 3 high bytes of the destination register are cleared. An lw would copy all 32 bits from the source. The high 3 bytes in the receiver control register are probably read as zeros, but lbu is a safer bet.

- The *receiver control register* resides at memory address 0xffff0000. Only bits 0 and 1 are used.

```
 31               0
+------------------+
|    unused    | I| R|
+------------------+
```

- Bit 0 (R) is the *ready* bit. It is set to 1 by the keyboard controller when a key is pressed. It is cleared automatically when the receiver data register is read.
- Bit 1 (I) is the interrupt-enable bit. This bit should be set to 1 by software if keyboard interrupts are to be used.
- The ready bit in the receiver control register and the entire receiver data register are read-only for the CPU. Attempts to change their values (e.g. using sw or sb) have no effect.
- The code below demonstrates a simple *spin waiting* (also known as *busy waiting*) loop. A spin waiting loop does nothing but poll an I/O device until the device becomes "ready" (new input is received, or an output device is done processing previous output). As soon as the device is ready, the loop exits and the I/O transaction occurs.

```
 ISO_LF              = 10 # Line feed (newline)
 SYS_PRINT_CHAR      = 11


 #  Receiver control. 1 in bit 0 means new char has arrived. This bit
# is read-only, and resets to 0 when CONS_RECEIVER_DATA is read.
# 1 in bit 1 enables hardware interrupt at interrupt level 1.
# Interrupts must also be enabled in the coprocessor 0 status register.

 CONS_RECEIVER_CONTROL           = 0xffff0000
 CONS_RECEIVER_READY_MASK        = 0x00000001
 CONS_RECEIVER_DATA              = 0xffff0004

# Main body
            .text
main:

            # Spin-wait for key to be pressed
key_wait:
            lw     $ t0, CONS_RECEIVER_CONTROL
            andi   $ t0, $ t0, CONS_RECEIVER_READY_MASK # Isolate ready bit
            beqz   $ t0, key_wait

            # Read in new character from keyboard to low byte of $ a0
            # and clear other 3 bytes of $ a0
            lbu    $ a0, CONS_RECEIVER_DATA

            # Print character and newline
             li    $ v0, SYS_PRINT_CHAR
            syscall

             li    $ a0, ISO_LF
             li    $ v0, SYS_PRINT_CHAR
            syscall

            jr     $ ra
```

- Note that characters read from the keyboard this way are not automatically echoed to the terminal. Higher level I/O facilities (like SPIM syscall functions) contain code to echo characters as they are received.

- The advantage of spin-waiting is that it responds almost immediately when an I/O device is ready.

- The down side, of course, is that the CPU is completely consumed with polling the device until it is ready, and no useful work can be done until after the I/O transaction. With a device such as a keyboard, which will produce at best about 10 input events in a second, the keyboard may be polled millions of times between keystrokes, even on a slow CPU. Hence, the CPU spends almost all of its time finding out that there is no input available, and once every few million iterations finds something useful to do.

- An alternative to spin waiting is *periodic polling*. In a periodic polling scenario, an I/O device is polled at various points during the execution of some useful code. Imagine the spin waiting loop above with a large amount of other code inserted, and you have a basic form of periodic polling.

- With periodic polling, the CPU can spend most of its time doing useful work. However, it is difficult or impossible to ensure that the device is polled at exactly the right frequency.

- Since the working code likely contains conditionals, we cannot always predict exactly how long it will take to reach the next polling instruction. If it takes too long (the device is under-polled), I/O events could be missed. If the software over-polls (polls far more often than events actually occur), then a significant percentage of available CPU time may be spent on polling, and the amount of useful work being done is reduced.

- When using software polling, the systems programmer must strike a balance between CPU time used for polling and the CPU time used for other work.

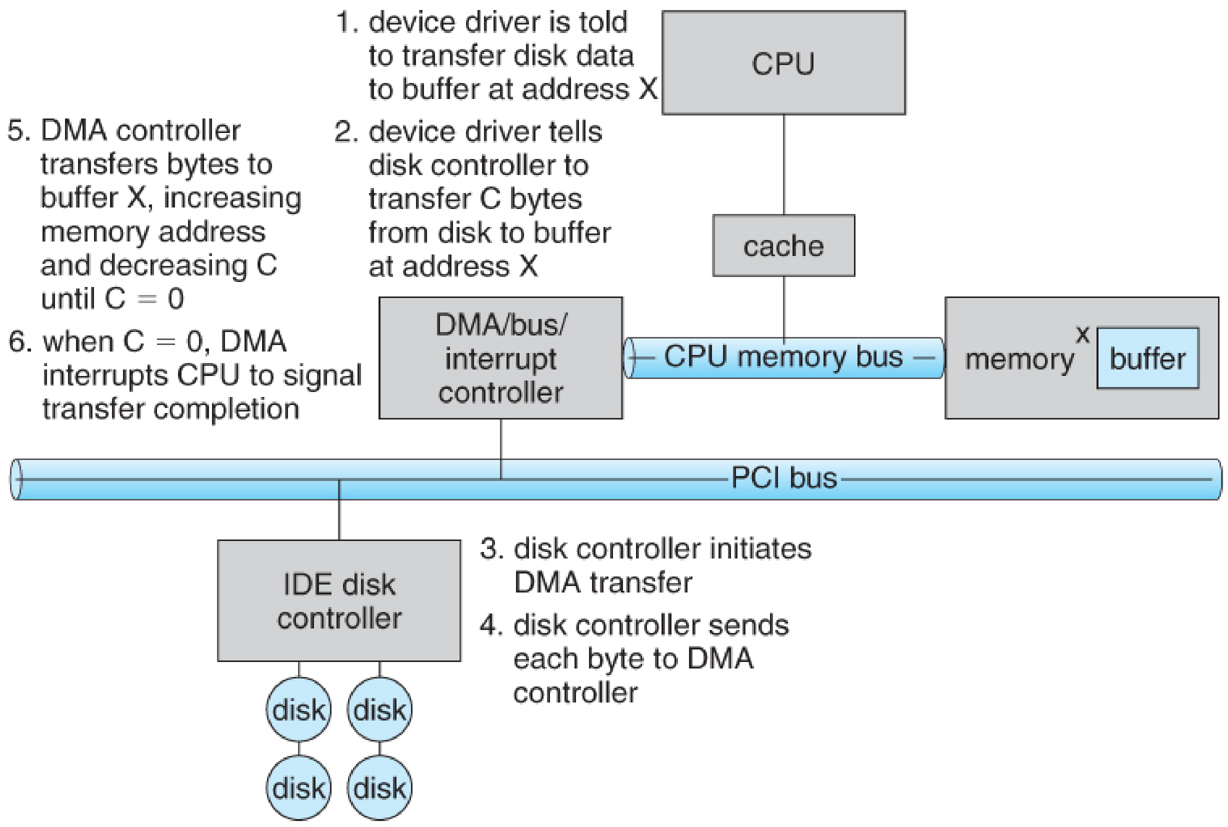
- If missing an I/O event would be unacceptable (as with keyboard input), then the device must be over-polled to ensure that this doesn't happen, and overall system performance will be reduced.

- If missing an event is not critical (as with an output device becoming ready), then under-polling may be used so that the CPU is available for more useful work.

Reference : Ch 8 (Memory Mapped I/O control),Britton

Direct Memory Access

- For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.
- Instead this work can be off-loaded to a special processor, known as the *Direct Memory Access, DMA, Controller.*
- The host issues a command to the DMA controller, indicating the location where the data is located, the location where the data is to be transferred to, and the number of bytes of data to transfer. The DMA controller handles the data transfer, and then interrupts the CPU when the transfer is complete.
- A simple DMA controller is a standard component in modern PCs, and many *bus-mastering* I/O cards contain their own DMA hardware.
- Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.
- While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.
- DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as *Direct Virtual Memory Access, DVMA,* and allows direct data transfer from one memory-mapped device to another without using the main memory chips.
- Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons.


1. device driver is told to transfer disk data to buffer at address X
CPU
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
cache
6. when C = 0, DMA interrupts CPU to signal transfer completion
DMA/bus/ interrupt controller
CPU memory bus
memory
x
buffer
PCI bus
IDE disk controller
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
disk
disk
disk
disk

The above figure illustrates the DMA process

References : Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Eighth Edition ", Chapter 13