14.3. Memory-Mapped I/O

See Britton chapter 8.

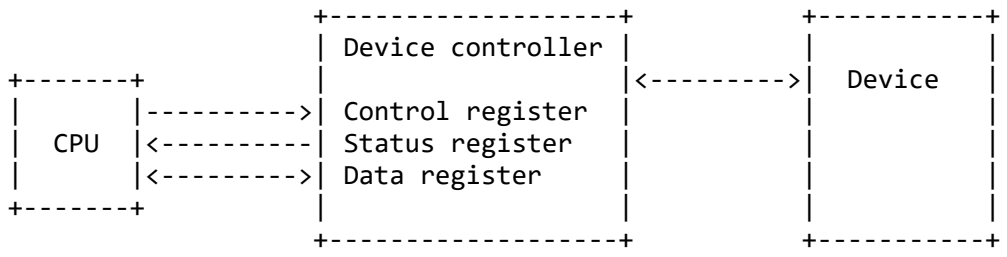
14.3.1. Introduction

From the CPU's perspective, an I/O device appears as a set of special-purpose registers, of three general types:

- Status registers provide status information to the CPU about the I/O device. These registers are often read-only, i.e. the CPU can only read their bits, and cannot change them.
- Configuration/control registers are used by the CPU to configure and control the device. Bits in these configuration registers may be write-only, so the CPU can alter them, but not read them back. Most bits in control registers can be both read and written.
- Data registers are used to read data from or send data to the I/O device.

In some instances, a given register may fit more than one of the above categories, e.g. some bits are used for configuration while other bits in the same register provide status information.

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*.

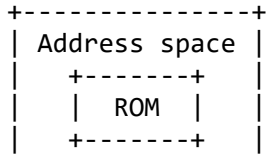


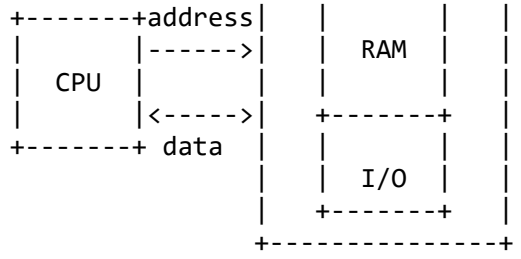
Simple devices such as keyboards and mice may be represented by only a few registers, while more complex ones such as disk drives and graphics adapters may have dozens.

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.





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.)

14.3.2. Memory-Mapped I/O with SPIM

The SPIM simulator provides simple keyboard and display devices. In order to access the keyboard and display controllers directly, instead of through syscalls, you must use the `-mapped_io` flags when starting SPIM.

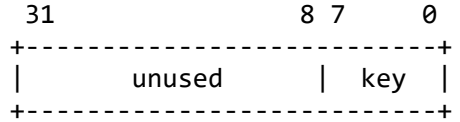
```
spim -mapped_io -file prog_with_drivers.mal
```

The presence of these simple device controllers is one of the big advantages of using a simulator to learn computer organization and assembly language. Developing real device drivers under operating systems like Unix or Windows is far too complex for an undergraduate course.

14.3.3. 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.



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. Interrupts are discussed in Chapter 15, *Exceptions*.

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.

The solution to the problems with spin waiting and periodic polling is offered by *interrupts*, which are discussed in Chapter 15, *Exceptions*.

14.3.4. Blocking vs. Non-blocking

I/O actions can be either *blocking* or *non-blocking*.

A blocking I/O action stops (blocks) the program from proceeding until the I/O transaction is completed. For example, a word processor will usually wait for a key press or a mouse click before doing any additional processing.

Blocking I/O is not appropriate for some applications, such as video games, for example. Imagine a video game that stops completely while waiting for the player to move the joystick or press a button.

With non-blocking I/O, a device is checked periodically (via polling or any other method). If no new input is available or the output device is not ready, the program continues. If new input is available or an output device has become ready, then the program may do something different, but it will continue to run whether or not an I/O event has occurred.

As an example, a getc function might be used to read a character from the keyboard. A blocking getc does not return until a key is pressed. It either spin-waits or sleeps until the user presses a key, and then inputs the character and returns it. A non-blocking getc function checks the keyboard *once* and either returns the key that was pressed, or some code to indicate than no new input was available.

14.3.5. Communicating with a Display Controller

The display controller works much like the keyboard controller.

The transmitter control register resides at address 0xffff0008. Bit 0 is the ready bit. A value of 1 in the ready bit indicates that the display is ready to receive another character.

A character is sent to the display by writing it to the transmitter data register at address 0xffff000c.

The ready bit is cleared automatically each time a write to the transmitter data register is performed. When the display has finished processing the character, the display controller sets the ready bit back to 1.

Caution

Programs must not write to transmitter data unless ready bit is 1. Doing so will produce undefined results.

As with the receiver control register, the ready bit in the transmitter control register is read-only. The transmitter data register is write-only. Data read back from it is not valid, and may not match what was last written.

The APE editor includes a “Low-level I/O” macro defining constants for direct access to the keyboard and display controllers.

14.3.6. Case-study: The PIC 18f8520 Microcontroller

The PIC 18f8520 microcontroller is a medium-scale processor used in the Vex robotics controller. The 8520's 10-bit analog-to-digital (A/D) converter module provides a fairly simple real-world example of a device controller, which is represented by 5 8-bit registers.

The A/D module samples an analog input signal (voltage) upon request from a program, and converts it to a 10-bit unsigned binary number.

Details can be found in the PIC data sheet at:

http://ww1.microchip.com/downloads/en/devicedoc/39609b.pdf

14.3.7. A Real-Time Clock

Not covered. Not possible with SPIM on a multiuser system.

14.3.8. Homework

Prev | Up | Next
14.2. I/O Devices | Home | 14.4. Disk I/O