Lecture 4

Input/Output and Interfacing

4.1 Introduction

Lectures 2 and 3 have described how a computer's CPU and memory function individually and interact with one another. But as yet no consideration has been given to methods of getting information in to or out of the computer — without which the whole exercise is pointless.

Computer Input-Output (I/O) can be described at a number of levels. Layers of software protocols build on yet more layers of hardware definition, with the aim, of course, of isolating those working at one level from (i) having to know detail of what happens below and (ii) second-guess how the level above will be used. You've seen modularity at work elsewhere to reduce the risk of design error.

Our first aim here is to consider I/O the lowest level in terms of register transfers. We then move on to consider typical I/O devices in the context of microcontrollers.

4.2 Register transfers — yet again

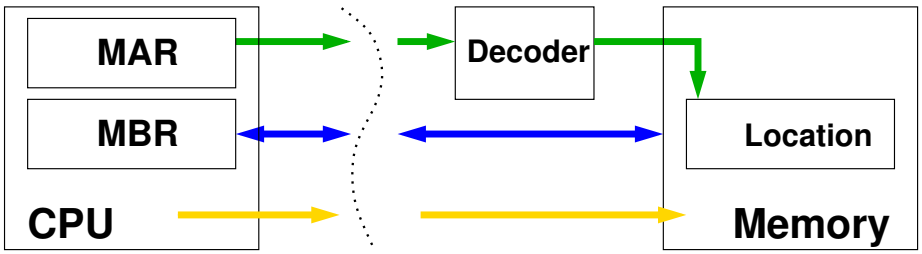
Much of the CPU and memory is nothing more than an elaborate register transfer machine, able to shunt Bytes from one place to another. It will then come as no surprise that I/O, yet again, involves register transfers.

Recall that the CPU is able to output to and input from *memory* using

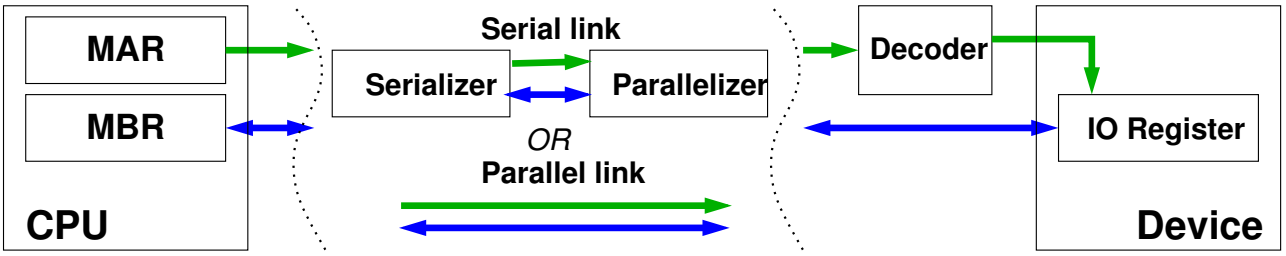
- an address bus and decoder to select a *particular register* in memory,
- a data bus to transfer the register's contents in or out of the CPU, and
- a control bus to carry signals such at Read, Write, and Output Enable.

The same idea of can be exploited to select a *particular device* for output or input.

The idea is sketched in Fig. 4.1(b), which also illustrates that while reading and writing data to memory is always performed in parallel, I/O to devices can be either parallel or serial. The former is, of course, has higher bandwidth, but the latter is necessary if the distance between CPU and device substantial. For many years, such serial communications were painfully slow, but fast serial links are nowadays available over the Universal Serial Bus (USB) and IEEE 1394 Firewire.



(a)



(b)

Figure 4.1: (a) A CPU selecting and transferring data to a particular register in memory shares much in common with (b) a CPU doing the same with a particular device.

4.3 Buses

In a desktop PC, the bus typically consists of 50 to 100 separate lines in the three functional groups of address, data and control, and made accessible on the computer's motherboard via slots. The example shown in Fig. 4.2(b) is an ISA bus (now obsolete, but chosen for the clarity of the picture).

A microcontroller will have the same buses, but there is no equivalent motherboard — buses, memory and i/o devices are usually held within a single IC package.

The design of buses is made awkward by the intricacies of timing. Not only do different devices have different speeds of operation, but transport delay accumulates from

- logic or propagation delay — the time spent between input and output changes in a gate, say 3–6 ns.
- capacitive delay — around $0.08\,\mathrm{ns\,pF^{-1}}$, with delays of 5 ns typical.
- transit time — $\sqrt{LC}$ where L and C are the inductance and capacitance per unit length — is around $6\,\mathrm{ns\,m^{-1}}$.

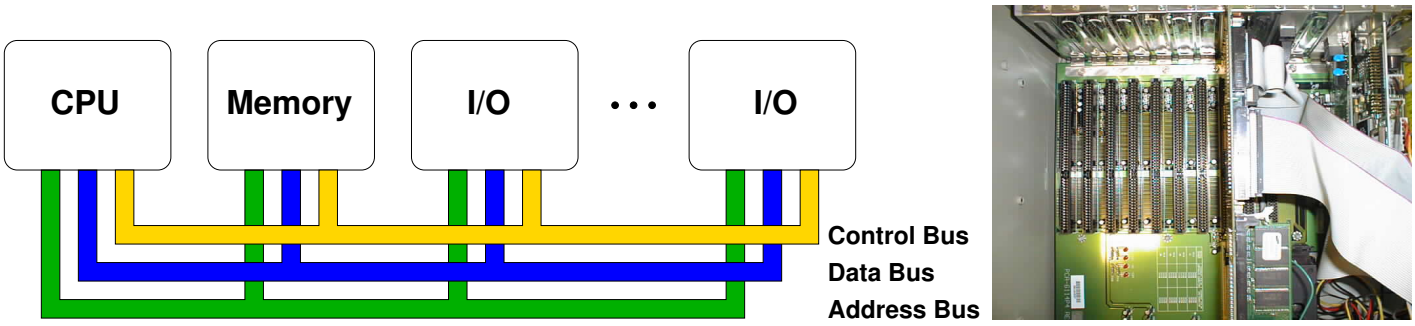


Figure 4.2: (a) Address, Control and Data buses schematically. In many computers the buses are made accessible using connectors into which cards are slotted.

There are two major design approaches to bus timing. Synchronous design requires devices to respond within a specified time with no continuous checks of whether the device did receive the data. This is fast, but not suitable when a mix of fast and slow devices sit on the bus. In asynchronous design, devices are considered to have their own clocks, and they a treated as separate RTL modules. The CPU and IO device then has to negotiate transmission and receipt using two control bus signals that continually monitor readiness to send and received data.

(It is rather dry material, but if you are interested there is a further note on the course website.)

4.4 Strategies for I/O

There are a number of generic types of I/O. An attempt to categorize those we will discuss is shown in in Fig. 4.3. Here we are primarily concerned with I/O that directly involves the CPU accessing registers on a device. This is called **Programmed I/O**, as opposed to Direct Memory Access and Channel I/O that will be mentioned in passing later.

Within programmed I/O we will compare and contrast **Port-mapped I/O** with **Memory-mapped I/O**, and will conclude that they are actually very similar, but that Port-mapped is more suited to microcontrollers.

Last we will contrast **Polled I/O** with **Interrupt-driven I/O**, the latter being a method of allowing devices to initiate I/O and hence avoid the waste of CPU cycles in the former.

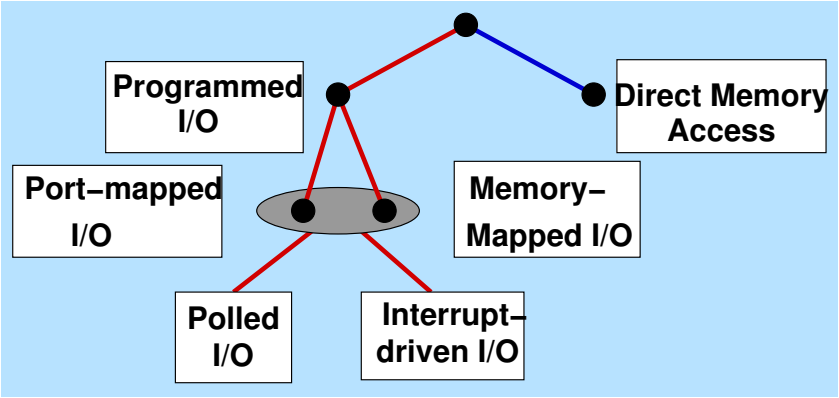


Figure 4.3: Some types of I/O, in something of a hierarchy. Don't read this too rigidly – some might put Interrupt-drive as a fourth branch at the top-level.

4.5 Port-mapped I/O versus Memory-mapped I/O

4.5.1 Register-based I/O

Consider the set of 8 I/O registers that sit on a special port address bus and special port data bus, as shown in Fig. 4.4(a). Our special address bus would need just three lines, and a 3-8 line decoder would determine which register to transfer to or from.

In practice there is no need to group the registers together physically. As illustrated in Fig. 4.4(b), they can sit separately on the bus — here as registers belonging to three separate devices occupying three slots.

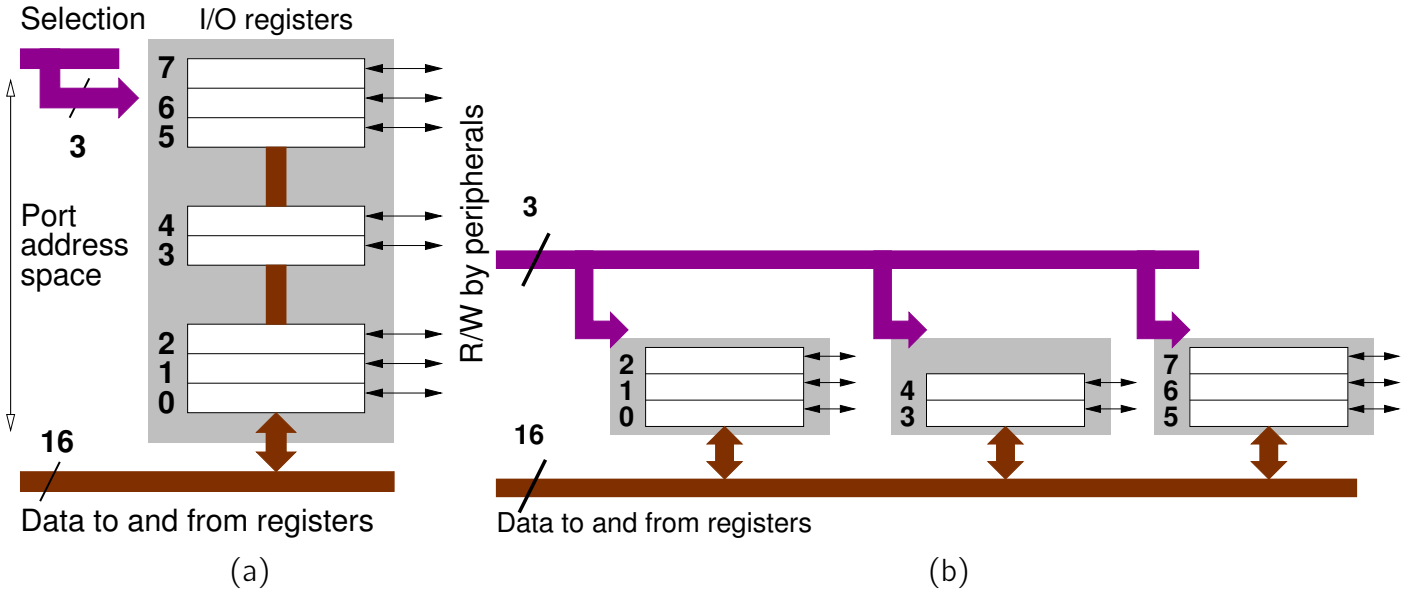


Figure 4.4: I/O registers on "special" buses.

4.5.2 Port-mapped I/O

A moment's consideration will suggest that the arrangement in Fig. 4.4 could be much more economically achieved by using the *main* data and address buses, and just connecting the lowest 3 address lines A0-A2 to the IO registers.

This nearly works. Unfortunately the addresses (0 to 7) of the IO registers would overlap those of the corresponding registers in main memory. This is easily solved by introducing a new control level to select between use of main memory and use of I/O register — we could call this USEmem/$\overline{\text{USEport}}$. On the CPU side, this would be provided by a Level from the Control Unit. On the Memory/Port side, USEmem will connect up (perhaps with some intermediate logic in a system with several memory chips) with the memory's ChipSelect, and we need a similar ChipSelect on each set of IO registers.

This approach to I/O, drawn out in Fig. 4.5 is called **Port-Mapped I/O**.

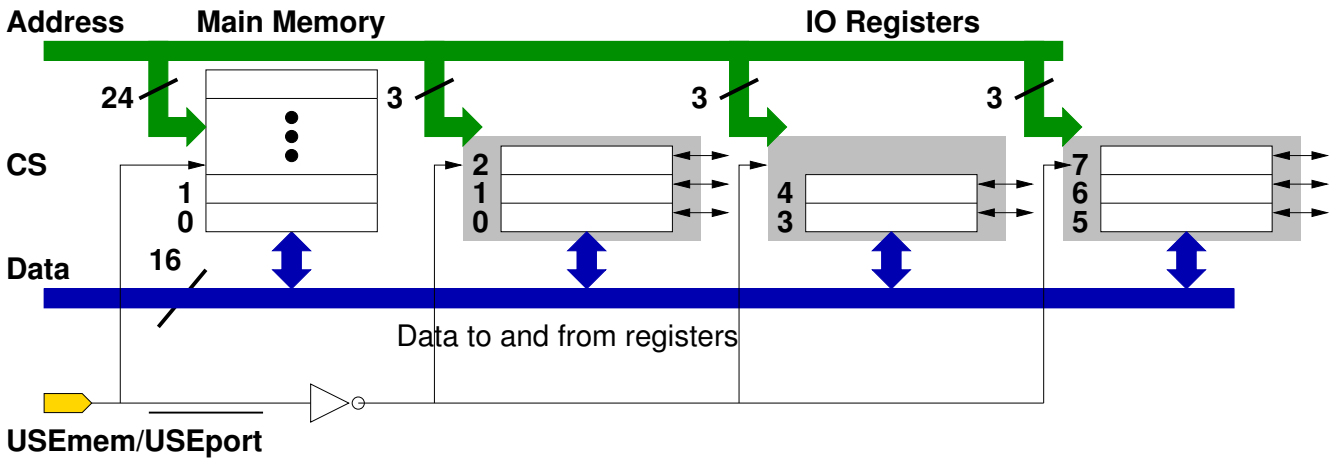


Figure 4.5: Port-mapped I/O registers can sit on the main buses, but there must be a control level that determines whether to use main memory or the ports.

A further requirement is that each IO register is readable and writeable **both** by the cpu *and* by the external peripheral. Such devices are called "dual-ported" registers or, when grouped in numbers, dual-ported RAM. One might imagine making a Dual-ported RAM from a Single-ported RAM with multiplexers on both the address and data lines. Commercial devices are much more sophisticated, allowing simultaneous reading from the same address, and simultaneous reading and writing from different addresses.

Usually a peripheral device will require several port addresses, some for the transfer of "proper" data, and others for the transfer of status information about the device. Status information is used *inter alia* to implement another level of handshaking (see later). In Fig. 4.6 we assume that 12 address lines (of our usual 24) are used to address 2^{12} ≡4K I/O registers, 2 Bytes wide.

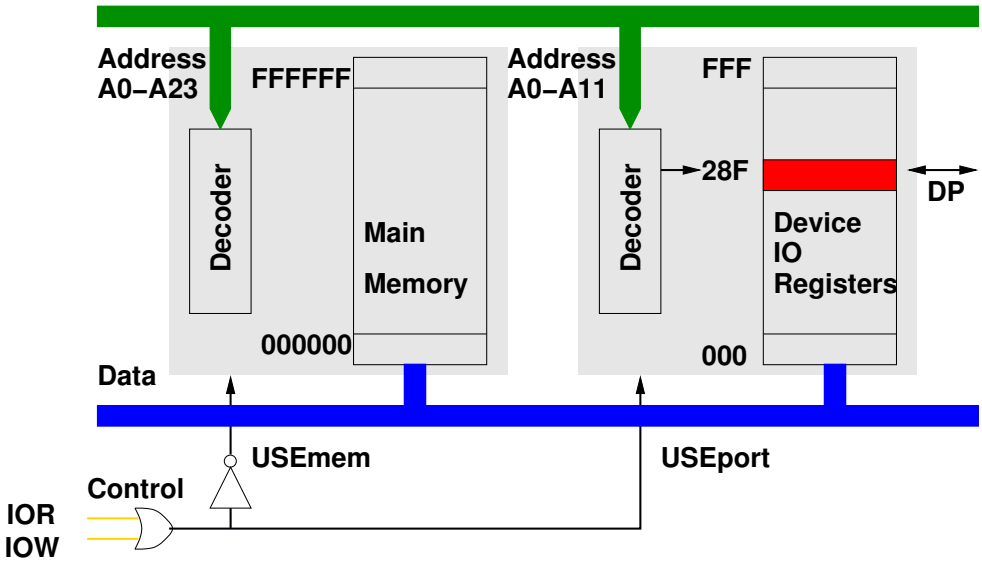


Figure 4.6: Port address space with 2^{12} locations 0 to FFF in hex.

CPU's that support port-mapping are equipped with `IN` and `OUT` instructions. For example:

```
LDA# 0x9A80   ;; get 9A80 hex into accumulator
OUT  0x28F    ;; and send it to port with address 0x28F
```

In Fig. 4.6, it is assumed that the `IN` and `OUT` instruction generate CS Levels IOR and IOW, respectively, so

$$\text{USEport} = \text{IOR} + \text{IOW}$$
$$\text{USEmem} = \overline{\text{USEport}}.$$

4.5.3 Memory-mapped I/O

Recall that it unlikely that the entire *memory* address space is filled with physical main memory. We could do away with the need to select between USEmem or USEport if we put the I/O Registers into available gaps in the memory address space. The scheme, called memory-mapped I/O is sketched in Fig.4.7.

As earlier, dual-ported registers are required, but now writing and reading to a device uses the standard instruction for loading from and storing into memory. For example:

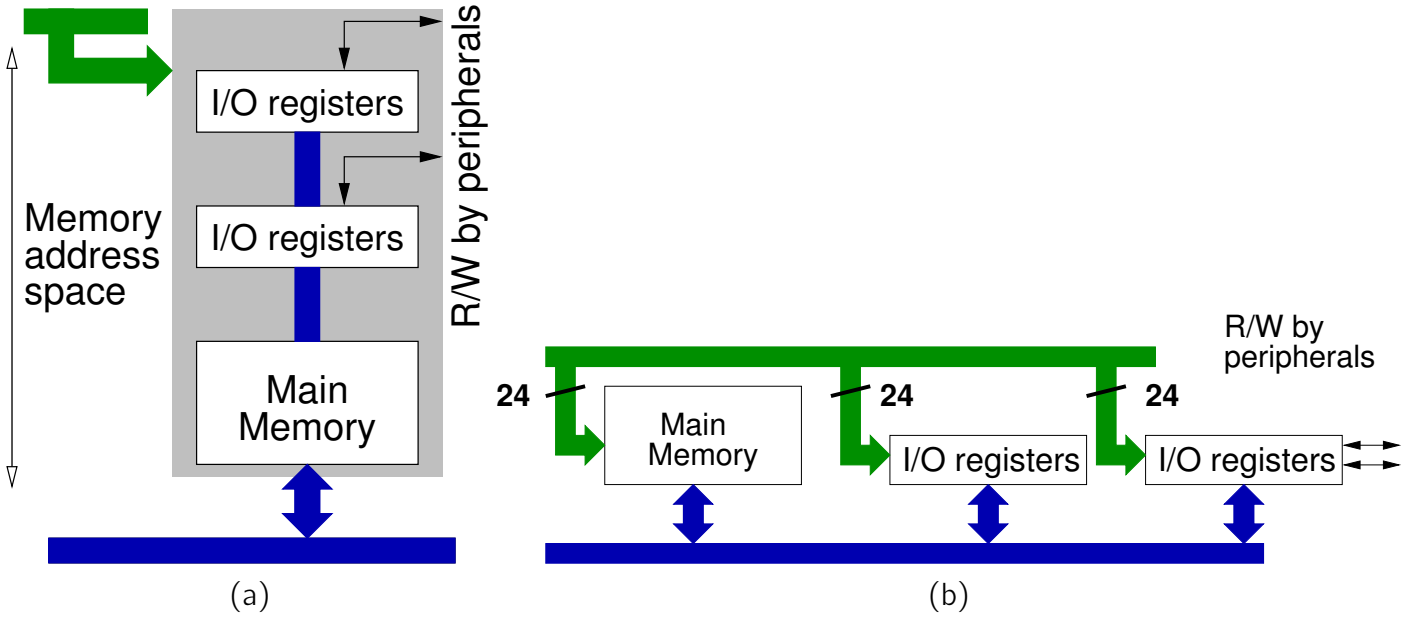


Figure 4.7: Memory mapped I/O. (a) IO-memory is in the memory address space. The physical layout in (b) indicates that io memory and main memory are physically distinct, and plug separately into the address and data buses.

```
LDA# 0x9A80    ;; get 9A80 hex into accumulator
STA  0x00CC00  ;; and send it to register at 00CC00 hex (24bit address)
```

4.5.4 The methods compared

Both methods have a mix of memory and I/O registers on the same bus — but port-mapped seems to have the disadvantage of requiring extra control wires.

So why is port-mapping used, particularly in microcontrollers? There are two main reasons.

- Microcontrollers typically have a relatively small address space (remember the 2^{18} figure). This can rather easily be filled with physical main memory, leaving no room for I/O registers.
- More crucial is the cost of decoding. In port-mapped I/O only the port address lines require decoding, whereas in memory-mapped I/O all the main-memory's address lines require decoding.

  For example, suppose we just have 8 I/O registers.

  – In port-mapped I/O, each IO device only has to decode 3 lines to determine whether it is being addressed, whereas

– In memory-mapped I/O all 24 have to be decoded. The jump from 3 to 24 does not sound much, but on the output side the difference is between 8 lines and 16 777 216 lines and gates. Ouch! (We exaggerate a little: there are cheaper ways of achieving the decoding, but it is a cost nonetheless.)

4.6 Scheduling I/O

4.6.1 Handshaking at different timescales

Full handshaking involve conversations between CPU and IO devices like ... **CPU:** "Are you ready?" **IOdev:** "I'm ready" **CPU:** "Have you got it?" **IOdev:** "I've got it".

We have already noted that handshaking occurs at the bus level to cope with subtle timing uncertainties in asynchronous buses. However this type of handshaking occurs via hardwired lines in the control bus, and at nano-second timescales.

We cannot reasonably use that fine-scale handshaking to cope with communication between the cpu and devices which may be many orders of magnitude slower. (This would akin to keeping a telephone open all day for a conversation that lasts 2 minutes.)

4.6.2 Buffered I/O

One approach to mitigating the mismatch in timescales and speeds is to output data in bursts, buffering it in fast memory on the slow device, as shown in Fig. 4.8. A First-In-First-Out (FIFO) buffer is filled quickly by the CPU, and then slowly emptied by the peripheral device.

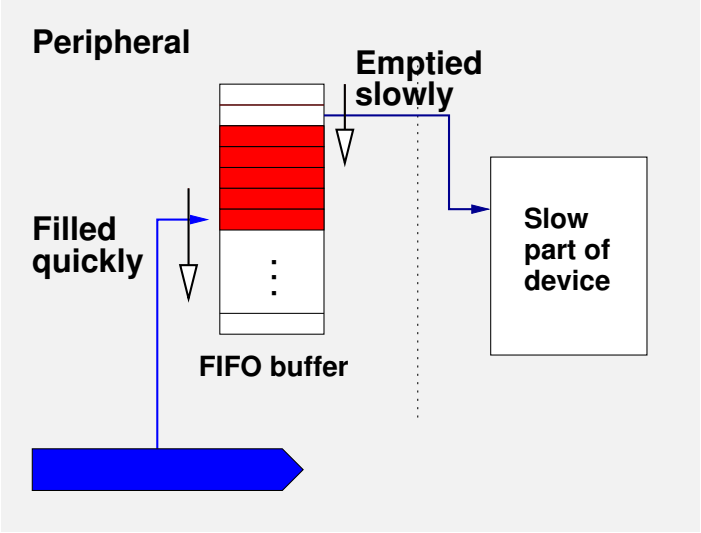


Figure 4.8: A fast FIFO data buffer.

Obviously, the buffer must be bigger than the volume of data that might be output in

one burst, but even this is not robust. To ensure no data is lost the IOdevice must, at very least, have a bit in its *device status word* that indicates to the CPU whether it is READY or NOTREADY to receive data.

This then would seem to require handshaking at, or at least involving, the I/O software level. We consider two ways of coordinating handshaking at this higher level, **Polling** and **Interrupts**.

4.6.3 Polling

Polling is a simple, but often simplistic, software solution to determine whether a device is ready to receive (or transmit) data. The CPU regularly check the status of the device by reading the status bit (or word, if there are several bits).

As an example, the following code transfers 100 word of data from a array in memory (starting at location 0x200) to a device whose status is checked by polling.

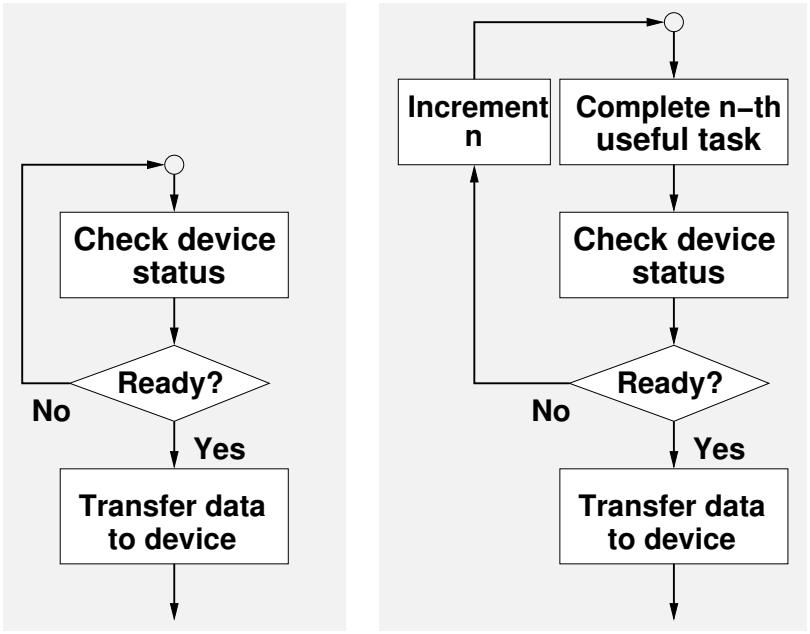
The output device is port-mapped at port 500 and its status bits are at location 501.

```
       LDA #200   ; Load base address into AC
       STA 22     ; Loc22 hold address of array element
LOOP:  IN  501    ; read device status word
       AND #1     ; is the lowest bit equal to 1?
       BZ  LOOP   ; if not, jump to LOOP
       LDA (22)   ; Load contents of array location
       OUT 500    ; write them to device
       LDA 22     ; Load and
       ADD #1     ;  increment the location
       STA 22     ;    and store it back
       SUB #300   ; Have we gone too far?
       BNZ LOOP   ; If not, carry on looping
```

However this is obviously inefficient. Three instructions at the core of the polling loop get executed over and over.

How wasteful could this be? Suppose the IO device handles some 1000 Bytes per second. To handle one Byte takes 10^{-3}s. A 2 GHz CPU taking 4 cycles per instruction will take $(3 * 4)/2 \times 10^9 \approx 10^{-8}$secs to handle compute the polling loop, and so the polling loop will execute approx 10^5 times while waiting for the device to be ready again!

Such inefficiency may be of no concern for an embedded microcontroller performing the simplest of tasks. For example, no-one cares that the microcontroller in a cash-dispenser is wasting its time waiting for your next button press — after all, what else would it do? (No, don't speculate.) However, if there is serious computation to be done in a real time system, it matters a great deal.



(a) (b)

Figure 4.9: (a) Basic polling. (b) An attempt to poll while getting on with other useful tasks.

One way of mitigating the gross waste of Fig. 4.9(a) is to attempt to do something useful between polls, as in Fig. 4.9(b). However, this relies on the program (and hence programmer) ensuring that a device is checked sufficiently often — a miserable task in a system where the various useful tasks take different times, and where there are several devices with differing I/O rates.

4.6.4 Interrupt-driven I/O

If software alone cannot solve the problem, we must return to hardware. Rather than one-to-monitoring, a better approach is to allow devices to signal in hardware to the CPU that they require attention.

This is the basis of **interrupt-driven I/O**, in which the peripheral literally interrupts the processor from its usual grind through the programmed instructions.

Elsewhere and already resident in the program memory are a set of **interrupt service subroutines**, each of which contains the instructions to handle a particular sort of interrupt. When the interrupt is detected (on the interrupt request (IRQ) control line), the processor stops executing its currently programmed instructions, jumps to execute the appropriate subroutine and, once completed, returns to carry on with the programmed instructions.

There are various things that need to be done when an interrupt is received.

1. Finish executing the current instruction.
2. "Recognize" the interrupt. Ie, determine which service routine is needed.
3. Save all the CPU register contents (PC, Registers, and Status Word) in memory. The Stack memory is used for this.
4. Jump to the routine, execute it, and return.
5. Restore the PC, registers and status word from the stack.
6. Continue with original program sequence, as if nothing had happened.

So this is mostly like jumping to a standard subroutine. However, as this is a routine which could be called at anytime and hence anywhere, there are no parameters to be passed. In addition, notice that the registers and status word are saved. When a programmer writes a subroutine, it is assumed that s/he will write it so that data being worked on is not lost. However, an interrupt routine is not called by the programmer, but by the *machine*, and can occur at any time.[1] The machine must make sure the CPU's state can be fully restored after the interrupt.

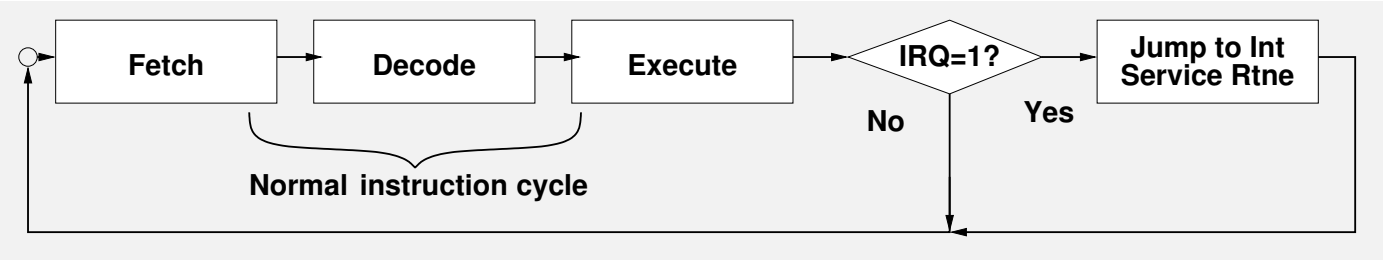


Figure 4.10:

Care has to be taken not to get deluged by interrupts. A devices interrupts with particular "priority" n, and when an interrupt at priority n is being dealt with, interrupts at priority $\leq n$ are blocked. This doesn't cancel the interrupt request, but simply prevents it getting through. However, the most urgent form of interrupt is "non-maskable". These are usually associated with system functions affecting the machine's well-being. (Further reading in Clements: Principles of Computer Hardware.)

[1] *Nobody* expects the Spanish Inquisition.

4.7 Microcontrollers

We are now in a position to appreciate the architecture of a typical microcontroller which integrates on a single chip

- CPU,
- Memory (ROM, RAM and (E)EPROM),
- IO Ports (digital and A/D, pulse, serial/parallel comms, etc),
- IO interrupt control,
- Timers, and
- Internal buses to connect the components.

You might like to think about the changes you would need to make to our BSA to use separate Program and Data Memories.

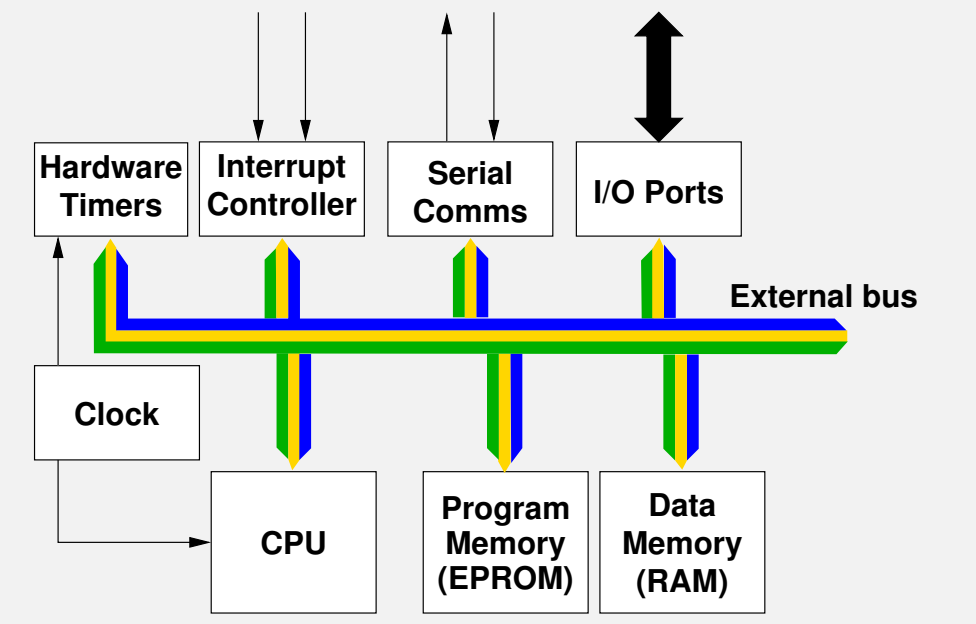


Figure 4.11: A microcontroller with a Harvard architecture, where program and data memory are separate (as used, eg, in the PIC family of micro-controllers).

Now we consider devices which might be connected to the I/O ports — not printers and the like (most texts will discuss in some detail devices such as keyboards, printers and disks), but devices used as part of a general engineering system for data acquisition and control output.

4.8 Interfacing for Real-time Control

4.8.1 Simple Digital Input

By "simple" digital input we mean input from switches or other devices that produce a few bits to indicate their state. For example, a car electronic managment system may wish to monitor 10 switches indicating whether 5 seats are occupied and 5 seat-belts buckled; or a chemical plant might wish to report whether valves are shut or not.

In Fig. 4.12 the 5 single wires are connected to the bits D[0] ... D[4] of the input register, and bits D[7:5] are grounded. The inputs are clocked in regularly. Using an `IN portaddress` instruction would transfer the register's contents to the AC.

Suppose the port has address 0xFA. Suppose all D[4:0] are meant to be high, and alarm is meant to be set off if any of the 5 inputs is low. The following (wasteful!) polling code would monitor the inputs.

```
again: IN 0xFA
       SUB #0x001F
       BNZ alarm
       JMP again
alarm: ..
```

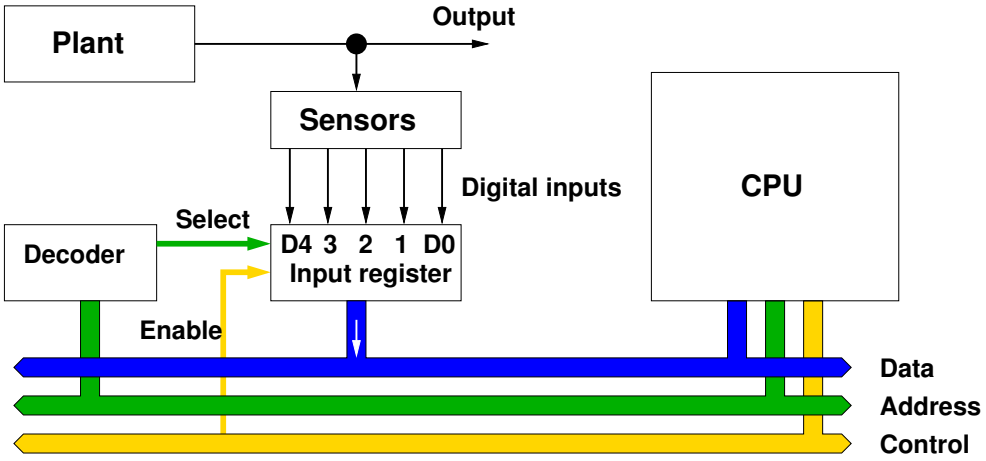


Figure 4.12: The plant's output is sensed and assumed to be turned into a binary (0/1) input.

In Fig. 4.12 the plant's output is sensed and assumed to be turned into a binary (0/1) input, which is clocked into a register. Note that as only the peripheral writes to the register, and only the CPU reads from the register, it does not even need to be Dual Ported! So a 74273 Octal D-type flip-flop with Clear would do. The CPU reads the register by putting its PortAddress onto the Address Bus, and clocking its contents into the MBR. The Enable line is derived from the IOR level mentioned earlier.

4.8.2 Simple Digital Output

In Fig. 4.13(a) the register is turned around to allow output from the AC to the output register.

In Fig. 4.13(b) a relay for switching a large current is controlled. The relay is connected to the msb output Q[7] of the flip-flop, but Q[6:0] are used as well.

Suppose the port is at OxFB, all output lines are uand that the current status is stored in memory location 0x0123. We want to send the entire current status to the register, except that we must ensure that the MSB is 1 to turn on the relay.

The code is

```
LDA 0x0123   ;; get desired status from memory
OR #0x80     ;; OR with binary 1000 0000
OUT 0xFB     ;; out to port
STA 0x0123   ;; store status for future use
```

OR'ing with 0x80 ensures that bit 7 is switched ON, but leaves the other outputs unchanged.

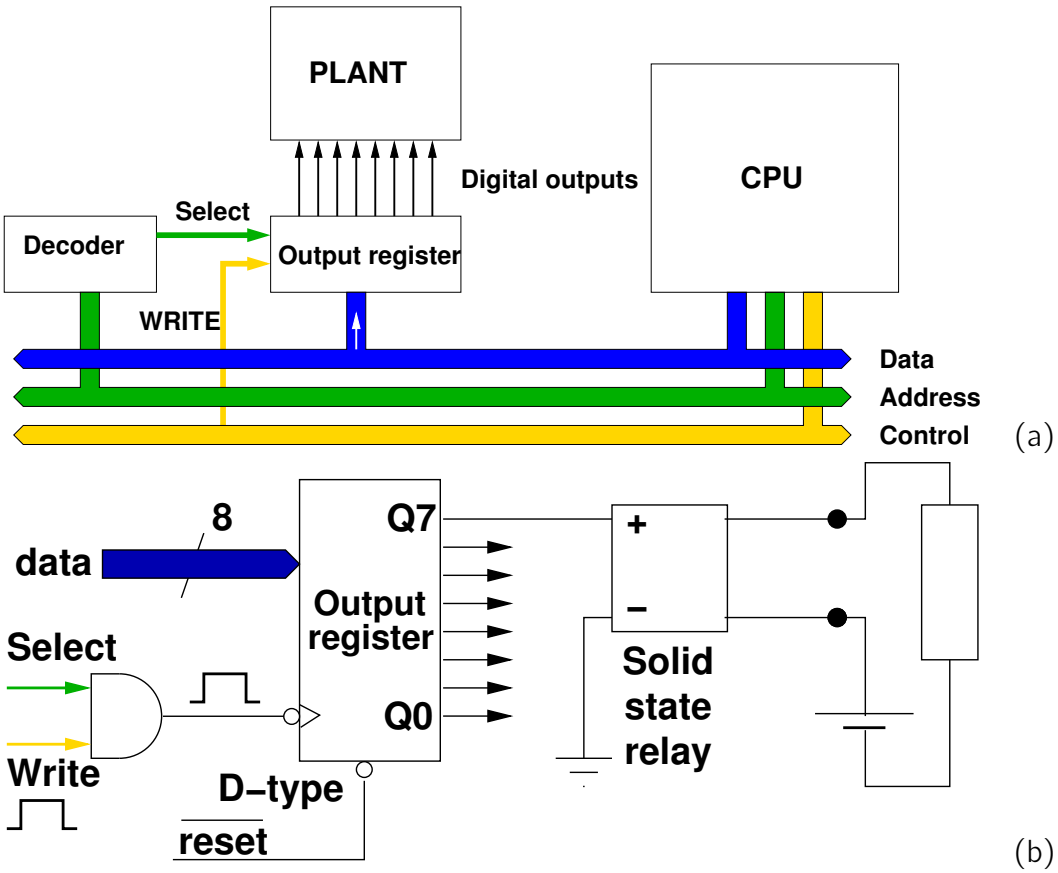


Figure 4.13: (a) Simple digital output. (b) Relay control for switching a large current.

4.8.3 Analogue voltage input

This would handle any analogue sensor input such as a thermistor, strain gauge, etc

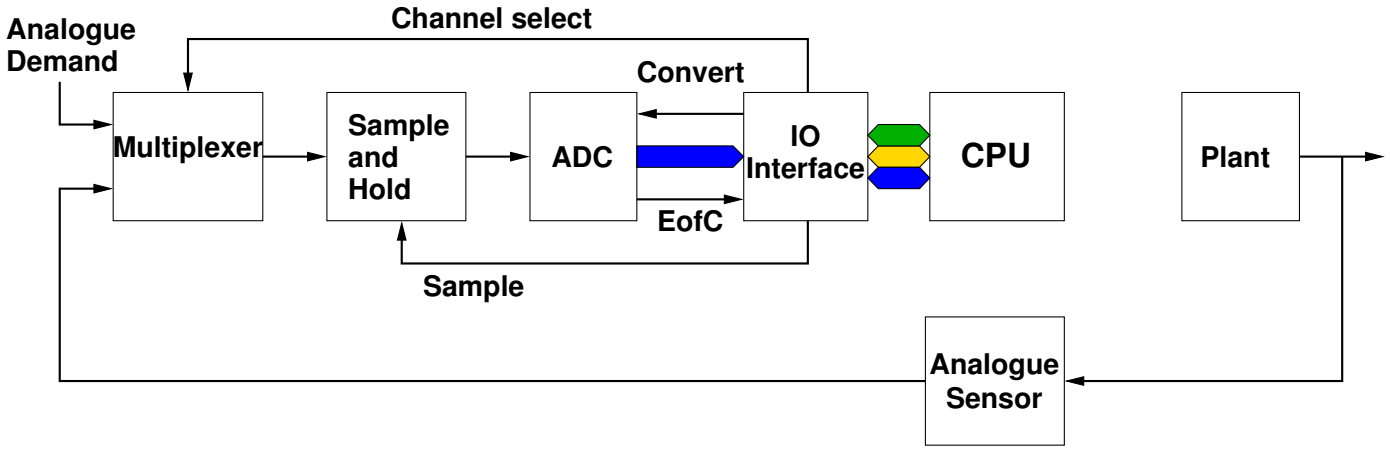


Figure 4.14:

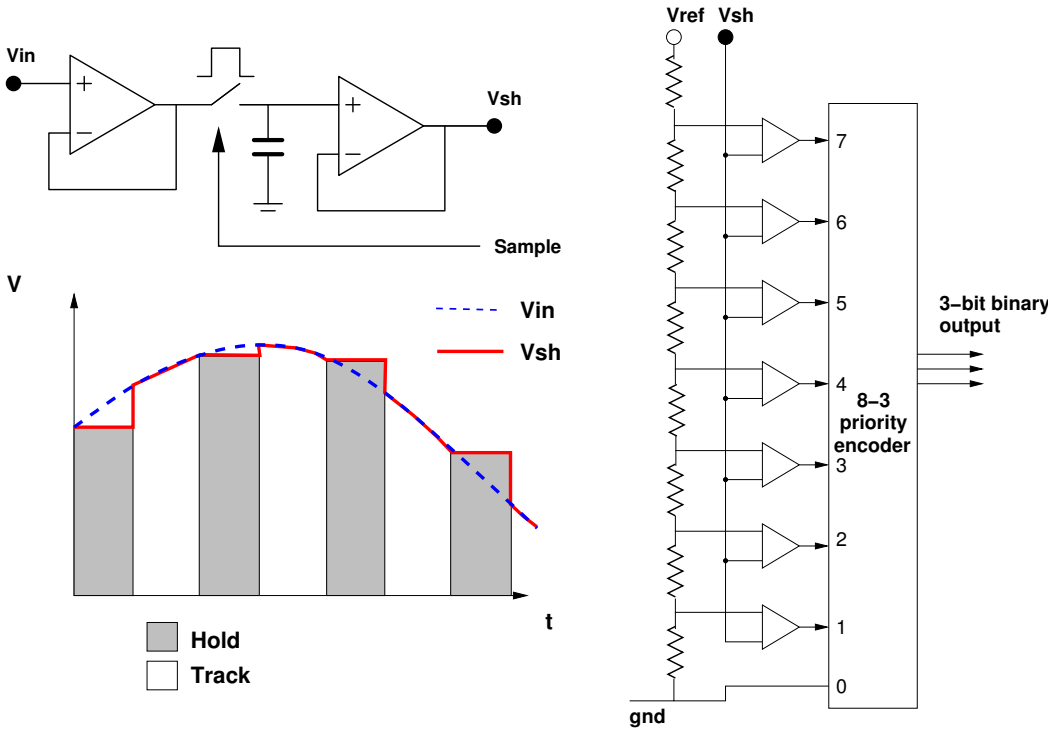


Figure 4.15:

4.8.4 Analogue voltage output

This of course requires a D to A converter, typically achieved using an R-2R ladder as described in you P2 OpAmp notes.

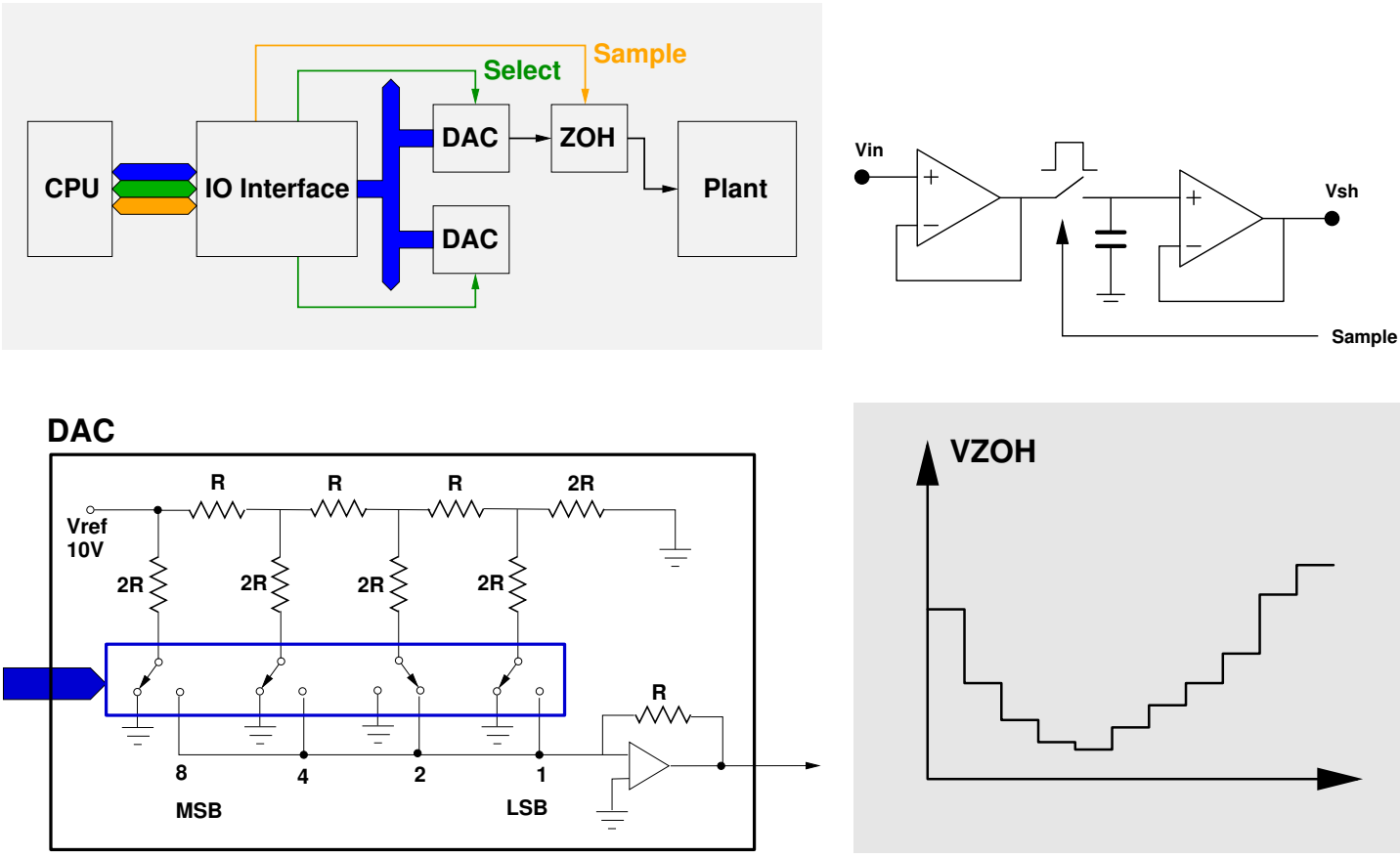


Figure 4.16: DAC interface, and R-2R ladder for D/A conversion.

The IO interface merely allows several DACs to be handled easily. The address will select both the Interface and the particular DAC, and the Digital o/p is written from the AC to the output port, and thence to the DAC. Within the DAC, the Data bits set the switches in an R-2R ladder. On the output side of the DAC is a Zero-order-hold, which is a sample an hold device which ensures the DAC analogue output stays at a fixed value until the next conversion.

4.9 Digital Control System

The devices for input and output have been introduced separately, but in **closed loop control** we want both present so that the sensor measurements can affect the output.

There is an endless list of applications — active suspension, robot positioning, building stabilization, chemical reaction control, air conditioning control, docking control for ships, and so on.

A typical representation of such a system is shown in Fig. 4.17, where the continuous output $y(t)$ is sampled and held and digitized, giving discrete time samples of the input $y(kT)$, where T is the sampling period and $k = 0, 1, 2, \ldots$.

The desired value of y is $r(kT)$, and a key quantity to consider is the error between r and the ouput y

$$e(k) = r(k) - y(k) \ .$$

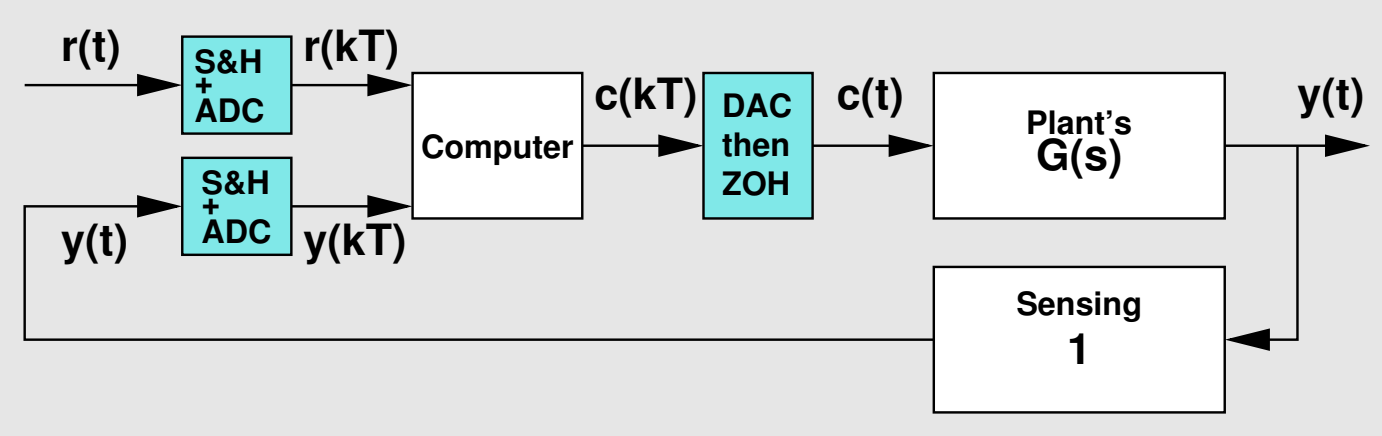


Figure 4.17:

Aside from this subtraction, what computations are done in the computer box?

That is the subject of the lectures on control, but, as a foretaste of heaven to come, consider in the time domain the commonly used (and abused) **PID controller**.

Its output is the sum of terms which are proportional to the error, the error's integral and the error's derivative.

$$c(t) = K\left(e(t) + \frac{1}{T_i}\int e(t)dt + T_d\dot{e}(t)\right)$$

Differentiate both sides to get rid of the integral

$$\dot{c}(t) = K\left(\dot{e}(t) + \frac{1}{T_i}e(t) + T_d\ddot{e}(t)\right)$$

Now substitute backward differences for the derivatives ...

$$\dot{c}(k) \approx \frac{c(k) - c(k-1)}{T} \quad \text{and} \quad \dot{e}(k) \approx \frac{e(k) - e(k-1)}{T}$$

and

$$\ddot{e}(k) \approx \frac{\dot{e}(k) - \dot{e}(k-1)}{T} \approx \frac{e(k) - 2e(k-1) + e(k-2)}{T^2}$$

Rearrange to obtain

$$\begin{aligned} c(k) &= c(k-1) + K\left[\left(1 + \frac{T}{T_i} + \frac{T_d}{T}\right)e(k) - \left(1 + 2\frac{T_d}{T}\right)e(k-1) \right. \\ &\qquad \left. + \frac{T_d}{T}e(k-2)\right] \\ &= c(k-1) + Ae(k) + Be(k-1) + Ce(k-2) \ . \end{aligned}$$

The constants depend on the problem in hand (as you will learn).

The important thing here is that there is causal recipe for updating the output, using stored values of

- the previous output $c(k-1)$, and
- the previous two values of the error $e(k-1)$, $e(k-2)$

and

- the current error $e(k)$

It would not take much to write the instructions ...