Motivation

Buses implement communication among processors or among processors and memories. Communication is the transfer of data among those components. For example, in a general-purpose processor reading or writing a memory is a common form of communication. In a general-purpose processor reading or writing a peripheral’s register is another common form of communication.

A bus consists of wires connecting two or more processors or memories. Note that each wire may be uni-directional, as rd/wr, enable, and addr, or bi-directional, as data. Also, note that a set of wires with the same function is typically drawn as a thick line (or a line with a small angled line drawn through it). addr and data each represent a set of wires; the addr wires transmit an address, while the data wires transmit data. The bus connects to "pins" of a processor (or memory). A pin is the actual conducting device (i.e., metal) on the periphery of a processor through which a signal is input to or output from the processor. When a processor is packaged as its own IC, there are actual pins extending from the package, designed to be plugged into the socket on a printed-circuit board. Today, however, a processor commonly co-exists on a single IC with other processors and memories. Such a processor does not have any actual pins on its periphery, but rather "pads" of metal in the IC. In fact, even for a processor packaged in its own IC, alternative packaging-techniques may use something other than pins for connections, such as small metallic balls. For consistency, though, we shall use the term pin in this chapter regardless of the packaging situation.

A bus must have an associated protocol describing the rules for transferring data over those wires. We deal primarily with low-level hardware protocols in this chapter, while higher-level protocols, like IP (Internet Protocol) can be built on top of these protocols, using a layered approach.

Interfacing with a general-purpose processor is extremely common. We describe three issues relating to such interfacing: addressing, interrupts, and direct memory access. When multiple processors attempt to access a single bus or memory simultaneously, resource contention exists. This chapter therefore describes several schemes for arbitrating among the contending processors.
Timing diagrams help us in clearly understanding clock signals. They are very crucial in the functioning of an embedded system because they help us to get an idea as to how the different components of the system work together in synchronization. Also, they enable us to incorporate the delays and other practical errors offered by the circuit components, hence, preventing any scope of malfunction. In this module, you will understand the functioning of the timing diagrams.

**Prerequisites**

The student must have a basic understanding of Digital Logic especially Sequential Logic. It is advised that you become completely comfortable with the mentioned logic before continuing. The student must read the module of microcontrollers before proceeding with this module. It is also advised to read all the previous modules in this course to get a deeper understanding.

**Suggested time**

4 hours

**Learning objectives**

By the end of this module, the student will be able to:

- have a basic idea of timing diagrams
- relate them to the processor designed previously
- analyze different I/O addressing
- understand the ability of behavioral modelling of the system through clock diagrams
- understand the necessity and working of interfaces.
Concepts

Timing diagrams

The hardware protocol is usually described by means of a timing diagram. A timing diagram represents the values of the control and data signals used as a function of time. The value of control lines, like rd/wr and enable, are described either as high or low, whereas the value of data lines, like addr and data, are described either as invalid (a single horizontal line) or valid (two horizontal lines). Let us consider the example of a simple processor and memory interface as shown in Fig 1 below.

![Processor-memory Interface Diagram](image)

Fig 11.1: A processor-memory interface

Figure 2 describes the protocol for reading a data from the memory using a timing diagram. We can infer the following from the timing diagram:

(i) The address line should contain the valid address of the memory to be read, at least tsetup seconds before the enable signal goes high. If this condition is not met, there may be a setup time violation.

(ii) Once the enable signal goes high, the correct data will be available on the data line after time duration of tread seconds.

(iii) The rd/wr control line must remain low throughout the read operation.
Figure 3 describes the protocol for writing data to the memory using a timing diagram. We can infer the following from the timing diagram:

(i) The address line should contain the valid address of the memory to be accessed, at least $t_{setup}$ seconds before the enable signal goes high. Moreover, the data line should also contain valid data at least $t_{setup}$ seconds before the enable signal goes high. If these conditions are not met simultaneously, there may be a setup time violation.

(ii) Once the enable signal goes high, the data will be written to the memory only after time duration of $t_{write}$ seconds.

(iii) The rd/wr control line must remain high throughout the write operation.
Note: In the above protocol, the enable line is active high, meaning that a 1 on the enable line results in data transfer. However control lines may also be active low, meaning that a 0 on the line results in data transfer. Such a control line is generally represented with bar above it (enable), or a single quote after it (enable’) or an underscore l after it (enable_l). In this module we will use the term ‘assert’ to indicate the setting the control line to its active value (1 for an active high line and 0 for an active low line) and the term "deassert" to mean setting the control line to its inactive value.

Events: A timing diagram is characterized by ‘events’. An event is defined as a change in the signal value. For example in Fig 11.2 the change in the value of the enable signal can be regarded as an event.

Microprocessor Interfacing

Microprocessor interfacing as the name suggests helps us determine how the various components of the microprocessor, namely the memory, the processor and the I/O devices, communicate with one another. A common example of interfacing is determining how an input port is accessed by the processor. There are 3 protocols that we will examine in detail: I/O addressing, interrupts and direct memory addressing.
I/O addressing deals with how a microprocessor communicates with its I/O devices. This is done either via Port based I/O or System based I/O.

The numerous pins of a microcontroller may be classified as control pins or I/O pins. The control pins are used for performing specific functions like, setting the threshold value for the ADC or feeding the clock input. The I/O pins on the other hand are used for reading input data or writing output data. The I/O pins are further organised into groups to form ports. Hence a port consists of N pins, where each pin corresponds to particular bit. In Port based I/O the data is read from or written to a port similar to the way a register is accessed by the microprocessor. Port based I/O is also known as parallel I/O. This is because all the pins of the port are accessed simultaneously.
Example: Writing to a port: Let us consider an 8-bit port named A. To set all bits of Port A to 1, we can simply write an instruction like: PINA = 255.

Reading from a port: To assign the value of say Port B, to a variable a, we can simply write the instruction,

```
int a;
a = PORTB;
```

Hence if the input is a series of 8 one's the integer value of 'a; would be 255.

**Note:** Most micro-controllers also allow the programmer to specify direction of each pin, that is whether it is to serve as an input PIN or an output PIN. This is done via the Data Direction Register (DDRx). Setting a particular pin of DDR to 1 would configure it as an output port, whereas setting it to 0 would configure it as an input port.

In contrast to a port, a **system bus** is a set of pins consisting of address pins, data pins, and control pins. In this case the protocol for accessing the data is built into the hardware itself. Hence a single of instruction is sufficient to cause the read/write operations to take place.

![Fig. 11.5 System bus architecture](image-url)