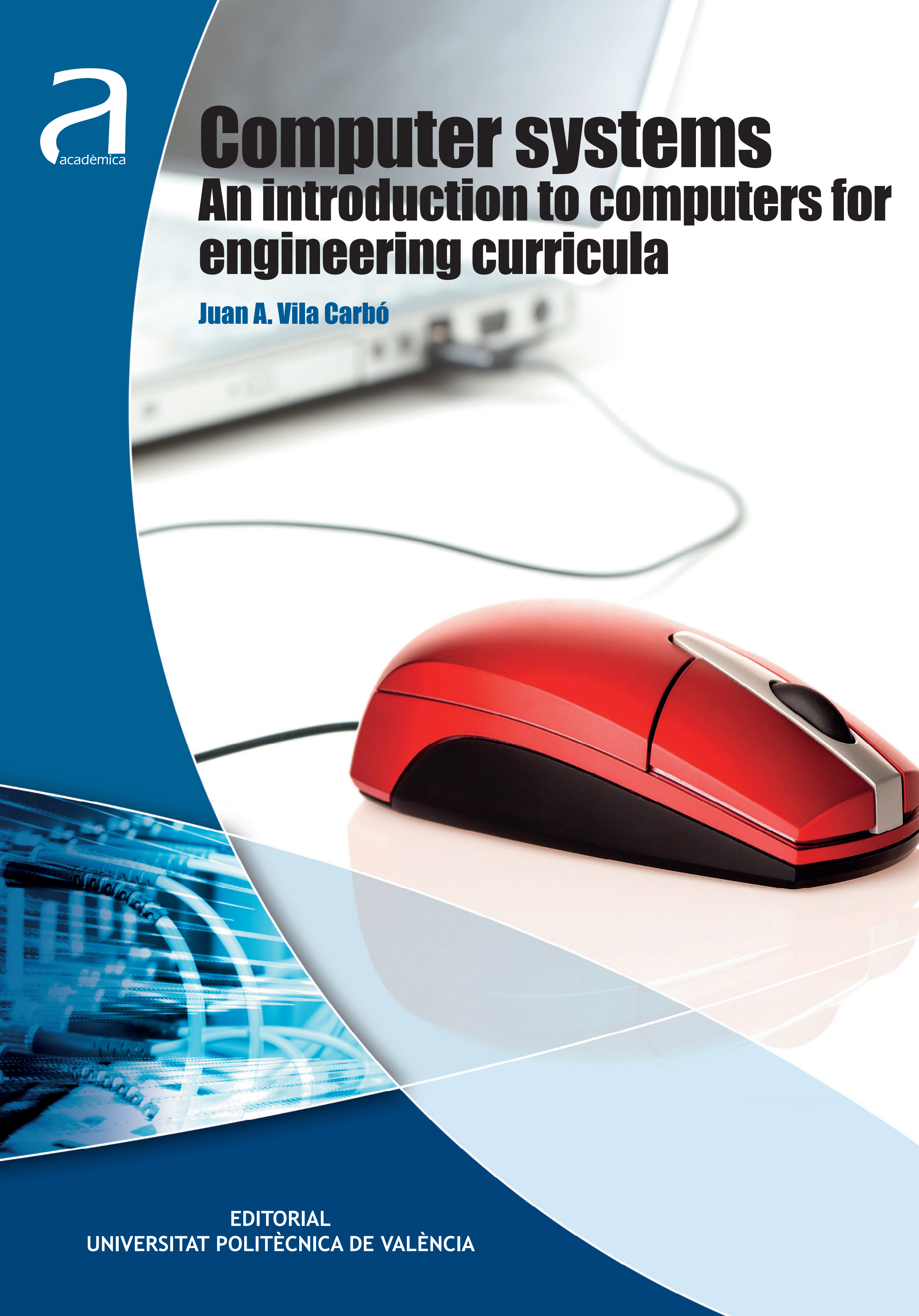
acadèmica

Computer systems
An introduction to computers for engineering curricula

Juan A. Vila Carbó



EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Computer systems

An introduction to computers for engineering curricula

Juan A. Vila Carbó

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Reference Book Series

How to cite this book: Vila Carbó, J.A. (2017). *Computer Systems: an introduction to computers for engineering curricula*. Valencia: Editorial Universitat Politècnica de València

The contents of this publication have been evaluated by a double –blind system as can be consulted on the following page
http://www.upv.es/entidades/AEUPV/info/891747normalc.html

Printed by: Byprint Percom, sl

ISBN: 978-84-9048-627-6
Print on demand

Abstract

Computers have definitely changed the teaching of engineering and also the professional practice: old design methods have evolved to computer based methods, numerical or finite elements methods are preferred to analytical methods, and the computer has become the main tool for an engineer. All engineering curricula have introduced computers as a basic subject. However there is not a consensus about the contents of that subject. There is a common agreement that basic programming should be introduced in all curricula and students should be also acquainted with the use the computer as a design tool for engineering. However, some engineering curricula need a deeper insight of computers since computer based systems (so called embedded systems) are also the goal of many engineering projects. This the case, for example, of aeronautical engineering, where aircraft navigation and control systems are really computers. It is also particularly true for telecommunication systems and industrial control systems which more and more rely on computer based systems. This book is thought as an introduction to computers for those who will study computer based systems in further subjects.

Although not strictly required, some notions about structured programming will help the reader to get the maximum from the book. In practice, we teach basic programming in parallel with this course. Programs are introduced in the book from the very beginning to formalize system's behavior unambiguously, to illustrate algorithms, or to simulate electronics. Programs are also used to fully understand computer concepts: for example, an interpreter or a Turing Machine can be better understood if we implement some basic version in software. I try to avoid in the book the arbitrary barrier between hardware and software that lies behind some books. Almost everything that can be done in software can also be done in hardware. It is a matter of efficiency to choose some given combination

of hardware and software in a given implementation. I try to stress this idea throughout the book.

Finally, to which extent should an engineer study the computer fundamentals? It would be probably hard to reach a common agreement about that. This book is an attempt to answer this question based on my own experience. An engineer is not a computer scientist or a programmer, but he has to be capable to work jointly with programmers, and to devise solutions which are suitable to be implemented using computer systems or computer programs. Engineering designs are becoming more and more complex and powerful, partly due to the use of computers. Some computer concepts, like abstract machines, system decomposition, system specification, system implementation, ... can be extrapolated to deal with the complexity of engineering designs. The book emphasizes these ideas too.

I hope the reader gets really motivated by the different topics introduced in this book and gets a conceptual view of what a computer is and how it works. This will enable him to further readings on computer systems and applications.

Juan A. Vila Carbó

jvila@upvnet.upv.es

Acknowledgements

I want to acknowledge Enrique Hernández Orallo, who started lecturing with me this course in the Aerospace curricula at Universitat Politècnica de València, for all his suggestions over this time and for his careful reading of the document. I also want to thank Àngel Perles Ivars for helping me to improve the document.

Contents

Chapter 1

Systems

This chapter introduces computers and their applications. Computers are complex systems. The main goal of the chapter is to provide some key concepts to address the study of complex systems: abstraction, black box, system and system composition. Different types of systems, from simple systems, to automata, and computers are progressively presented. Computers are introduced from two different perspectives: programmable machines, and processing information systems. Finally, a classification of computers as general purpose computers and embedded systems is presented and illustrated with some examples.

1.1 Black boxes and abstraction

Computers are complex systems: they are composed of a large number of components with a big number of specific features that defines each component. Moreover, all these components are interconnected and depend on each other. The overall design comprising all the components usually has a sophisticated behavior. This does not only apply to computers; machines, vehicles and other engineering designs are also complex. Being able to understand and to design such systems requires a structured way of thinking. The ideas of abstraction and black box are a good tool to face complexity.

A *black box* is an object which interacts with the environment through input and output signals (see figure 1.1). A black box can be described solely in terms of its inputs, its outputs, and how outputs relate to inputs (transfer characteristics). The key idea is that the implementation or internals of a black box are "opaque" (black), so there's no need to know how it internally works to be able to use it. In other words, we are rather interested in *what* a black box does rather than in *how* it does it.

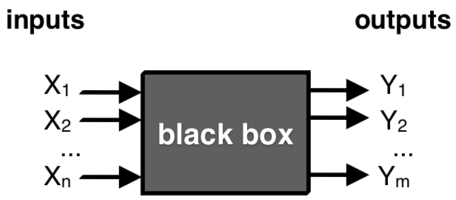


Figure 1.1: Black box concept. Black boxes can be described solely in terms of its inputs and its outputs. The implementation is “opaque” .

An example of a black box can be an “adder” object. An *adder* can be defined as a black box that takes two inputs and produces an output which is the arithmetic sum of the two inputs (see figure 1.2).

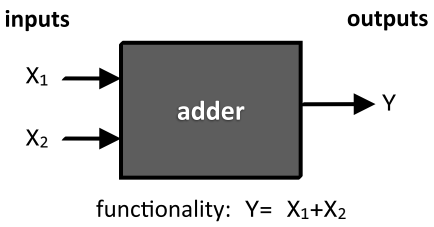


Figure 1.2: An adder. An adder is a black box with two inputs and an output which is defined as he arithmetic the sum of the two inputs.

A more complex example of black box could be a *vending machine*. The vending machine could be modeled as a black box with two inputs and two outputs. The inputs are a button which produces a *product code* that identifies the selected product, and a *coin* slot for the payment of the selected product. The outputs are the *product* dispenser and the output with the *change* of the payment (see figure 1.3). The vending machine could be implemented in different ways, using different technologies. But from an abstract point of view the internal mechanisms used to implement the machine are completely irrelevant. All we require to some possible implementation of the vending machine is to meet the machine specifications.

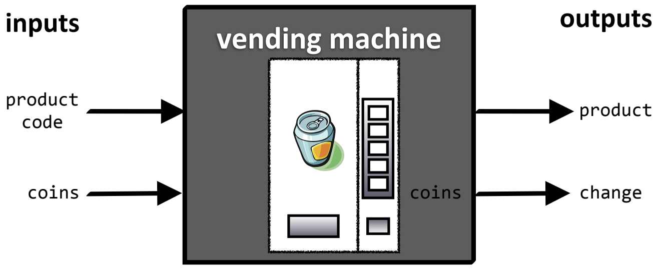


Figure 1.3: A vending machine.. This machine is as a black box with two inputs (*product code*, *coins*) and two outputs (*product*, *change*).

The *machine specifications* must clearly define the set of inputs and the set of outputs of the machine. These includes the set of all products, their prices, and the set of accepted coins. Additionally, they must also define the machine behavior, i.e, the sequence of possible operations that the machine performs: select a product, put coins, dispense the selected product, and give the change back. Specifications should also define "abnormal" conditions: product out of stock, not enough coins for the selected product.

Black boxes are a way of abstracting a real system. *Abstraction* is a process that allows to suppress all unneeded details from an object and to retain only information which is relevant for a particular purpose. This concept plays a central role in computer science and engineering. There is a famous quotation by C.A.R. Hoare, a remarkable computer scientists, stating that *"In the development of the understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction"* (Hoare 1972).

Let's use the example of a car for illustrating this concept: a car is a complex object with thousands of components and details, but for navigational purposes, i.e. to know or to predict the car's position, a car can be modeled as a point of mass with the following relevant attributes: its mass, its position in space, its velocity, and its acceleration. Things like the color of the car or the radius of the wheels are not relevant for navigation purposes. Abstraction allows to proper modeling an object by choosing the relevant attributes for some application while abstracting the non interesting ones.

Abstraction is also used to refer that most of the times we will deal only with abstract entities, i.e, conceptual models which do not match physical entities necessarily.

1.2 Systems

The concept of system usually denotes a compound entity. A *system* could be defined as a black box whose implementation is a set of interacting or interdependent components with some given structure and interconnections between components. Each component, in turn, can be also modeled as a system, so this becomes a recursive definition. The most important property about a system is that it meets its specifications and it provides added functionalities over the set of components.

In a system we may also be interested in specifying its internal structure as a set of interconnected subsystems. Those details were hidden in a black box. But even when we consider the internal structure of a system, it is still very important to distinguish between its *specification* and its *implementation*.

A system *specification* defines its *interface* and the definition of its transfer logic or functionality. The system interface comprises the definition of all the inputs and the outputs an their corresponding types. The type of an input or an output is the range of values that it may take. For example, the inputs and outputs of the adder object can be defined to range over the whole set of real numbers. It is a a continuous range. By contrast, the inputs and outputs of the vending machine are a set of *enumerated types*: they can only take a finite set of values. For instance, the *product_code* input may range over: $\{doughnut, sandwich, chocolate, lemonade, cola\}$. And similarly for the *coins* input: $\{10c, 20c, 50c, 1e, 2e\}$.

A system *implementation* defines its internal components and the system structure. It establishes some particular way of implementing the system specification using some well defined components and some way of interconnecting them. However, a system implementation is not unique: there can be different implementations of a system that meet the same specifications. This implementation can be done using different technologies: mechanical, electronic, ...

Validating a system consists of checking that a particular implementation meets the system specifications. Specifications usually refer to *correctness* conditions in terms of functionality: the machine performs as the specifications dictate. Two correct implementations may differ in their *performance*, for example their speed, weight, size, etc. Even more, not all “correct” implementations may properly work in hazardous or critical environments like industries or aircrafts.

Certifying a system refers to a process to check that it does not only meet the correctness specifications, but also some additional performance, quality and, mostly, safety requirements. There are usually certification procedures and standards from government agencies for products operating in safety critical environments.

1.2.1 Systems composition

One of the most important properties of systems is their ability to interact and to combine with other systems in order to configure a more complex system. Interfaces are key in this process, since they define the sets of inputs and outputs of a system that permit interaction across its boundary. An *open system* has an interface (or a set of interfaces) that allows it to interact with the environment (or different environments) and can be easily integrated or combined with other systems, possibly implemented by a third party. On the contrary, a *closed system* does not interact with the environment. This term is usually employed in a negative way to denote that a system cannot interoperate easily with other systems. Interoperability is often an important design criteria.

Systems can be interconnected by linking the outputs of a system to the inputs of another system. For example, the previously defined 2-operand adder can be used to build a 3-operand adder by combining two of them as shown in figure 1.4.

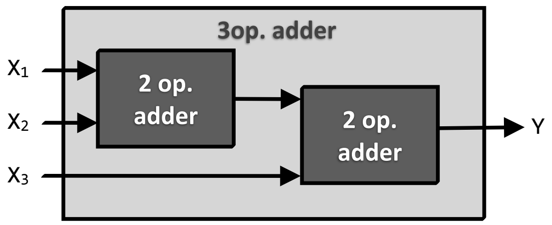


Figure 1.4: System composition. A 3-operand adder can be implemented as the composition of two 2-operand adders.

1.2.2 Continuous time and discrete-time systems

We will assume that the inputs and outputs of a system are, in general, time-dependent: $X_i = X_i(t)$, $Y_i = Y_i(t)$. This section analyzes the temporal properties of systems.

A *continuous-time* system is a system where inputs and outputs are defined over a continuous time domain. For example, we can define the inputs and outputs of the adder over the time interval $[0,5]$. If we represent the temporal behavior of the adder with a constant input $X_1(t) = 2$ and a variable input $X_2(t) = t$, we can see in figure 1.5 , that the output $Y(t) = 2 + t$ reflects at each instant of time t the sum of both inputs.

However, computer circuits and computer programs are “discrete-time” systems. A *discrete-time* system is a system where the time domain is a discrete set of values. An example of such domain could be $t = \{0,1,2,3,4,5\}$. In a discrete-time system inputs are sampled at some given time instants in the domain, while outputs are reevaluated each

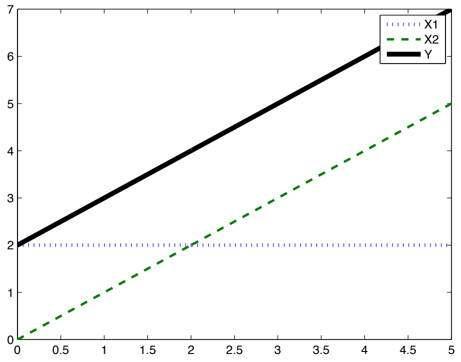


Figure 1.5: Continuous-time system. Response of the adder in continuous time.

time the inputs are sampled. Figure 1.6 shows the behavior of the adder for a discrete-time domain $t = \{0, 1, 2, 3, 4, 5\}$. Inputs may or may not be defined for instants not in the domain (for example for $t = 1.5$), but the system only samples its inputs on the time instants in the domain (shown as triangles in the figure 1.6). Another interesting question concerns the system response between two discrete-time instants. In principle this value is undefined, but most discrete-time systems keep the last value of the outputs (like in the dotted line of Y in figure 1.6) until they are reevaluated the next time instant.

Most computer-based systems and applications are discrete-time systems by nature. For example, consider a web-site that provides the exchange rate between currencies. The exchange rates are updated at finite intervals of time: for example every hour.

The *sampling period* is defined as the time difference between two consecutive samples. In general it is constant, but in some cases nonuniform sampling is also used. The sampling period for the adder example is constant and equal to 1 sec (sampling frequency is $1Hz = 1\ time/sec.$). A continuous-time system can be considered a system with an infinite sampling frequency.

Discrete-time signals (inputs and outputs) are typically written as a function of an integer index k. This way, notation $Y(k)$ (or Y_k) stands for the value of Y at discrete-time instant k, while $Y(k+1)$ (or Y_{k+1}) stands for the value of Y in the next discrete-time instant.

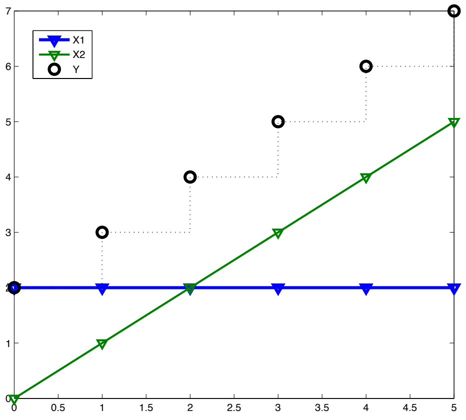


Figure 1.6: Discrete-time system..Response of the adder in discrete time with a sampling frequency of 1Hz.

1.2.3 Stateful and stateless systems

There are two main types of systems according to the nature of the transfer function that defines how outputs relate to inputs: stateful and stateless systems.

A *stateless system* is a system where the outputs at time t only depend on the inputs at that particular instant of time t. If we denote the set of inputs as $X(t) = \{X_i(t),\ i : 1\dots n\}$ and the set of outputs as $Y(t) = \{Y_j(t),\ j : 1\dots m\}$, it holds that:

$$Y(t) = \mathbf{f}\big(X(t)\big) \tag{1.1}$$

For example the adder object (in either of its versions: continuous or discrete-time) could be an example of such a system.

A *stateful system*, simply referred as a “system” most of the times, is a system whose outputs at time t depend not only on the current inputs at time t, but also on the history of inputs over time. In a discrete-time system this dependence of previous outputs can also be expressed in a recursive way as follows:

$$\begin{array}{lll} Y(k) = & \mathbf{f}\big(X(k), & Y(k-1)\big) \\ Y(k-1) = & \mathbf{f}\big(X(k-1), & Y(k-2)\big) \\ Y(k-2) = & \mathbf{f}\big(X(k-2), & Y(k-3)\big) \\ \ldots & & \end{array} \tag{1.2}$$

An example of a stateful system system could be an *accumulator*. The functionality of this system is to “accumulate”, i.e., to perform the summation of all previous inputs over time. The transfer function of this system could be expressed as:

$$Y(k) = X(k) + Y(k-1) \tag{1.3}$$

Figure 1.7 shows the temporal behavior of the accumulator for an example of input data.

k	$X(k)$	$Y(k) =$ $X(k)+Y(k-1)$
0	1	1
1	2	3
2	1	4
3	0	4
4	1	5
5	0	5

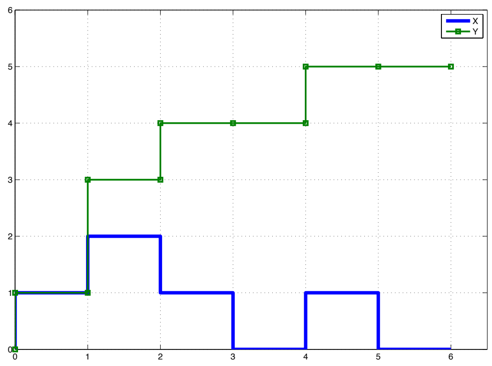


Figure 1.7: Accumulator. An example chronogram of the accumulator.

An example of a real accumulator could be the coin bank of a vending machine. This bank keeps the total amount of money collected in the machine over a period of time.

Stateful systems are defined by their *state*. On the contrary, stateless systems lack any state. The state of a system is the set of variables that describes enough the system in order to determine its future behavior. For example, in a vehicle, its position in space, its velocity, and its acceleration are typical state variables; knowing these variables, it is possible to determine the future system state. In the accumulator example, the state is the total amount of accumulated units. Knowing this value and the new input value, it can be determined the future state of the accumulator. Formally, we will define the system state as a function of the system outputs:

$$Z(k) = \mathbf{g}\big(Y(k)\big) \tag{1.4}$$

For example, the total amount of an accumulator can be defined as a state variable Z_1 whose value is $Z_1(t) = Y(t)$. We can also define another state variable Z_2 that indicates when "the accumulator is full". This variable a discrete variable that may only take two values $\{true, false\}$[1]. The predicate "the accumulator is full" will be said to be true if the total amount is ≥ 500. It will be false otherwise. We can express this function using a predicate $Z_2(t) = Y(t) \geq 500$.

Using the state concept, a stateful system can be redefined in a more general way as:

$$Y(k) = \mathbf{f}\big(X(k), Z(k-1)\big) \quad with \quad Z(k-1) = \mathbf{g}\big(Y(k-1)\big) \tag{1.5}$$

Keeping the system state requires using a new type of component in the system implementation that has the so called *memory* property. The memory property allows a system to store and to hold the value of a system variable over time.

A *register* is introduced as the most elementary stateful system, i.e, the most elementary system with the memory property. The register can be defined as a system that samples an input value and provides an output that matches this value until the next instant of time:

$$Y(k) = X(k-1) \tag{1.6}$$

Figure 1.8 shows a register and an example of its temporal behavior. The figure shows a *clock* signal which is a pulse that indicates the discrete instants of time where the signal has to be sampled. The register keeps the sampled value as its outputs until it is triggered again by a new CLK signal. The black thick line shows how the output is memorized until the next clock pulse despite the input changes.

Once the register example has been introduced, an accumulator can be easily constructed by combining a register and an adder as shown in figure 1.9. The solution consists of connecting the output of the adder to the register, so it can be "frozen" (or memorized). This frozen output serves as one of the inputs for the adder the next instant of time. Connecting one output to one input causes a loop in the system. This structure is known as a *feedback* system. To properly understand such a system you need to consider that the output of each subsystem is delayed (at least some infinitesimal amount of time) with

[1] This is usually called a boolean variable.

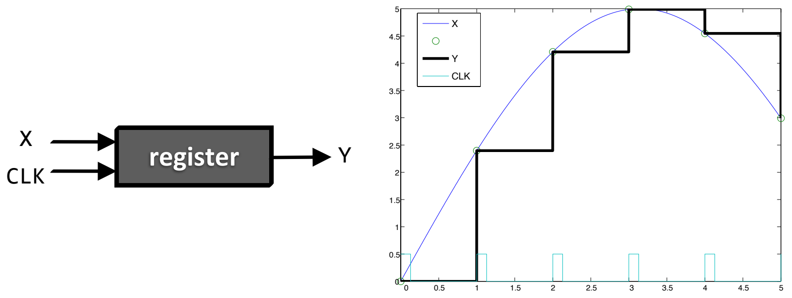


Figure 1.8: A register. A register scheme and chronogram. The chronogram shows the input, the output, and the clock signal.

respect its input. This way, the adder always gets the previous output of the accumulator (and not the current one).

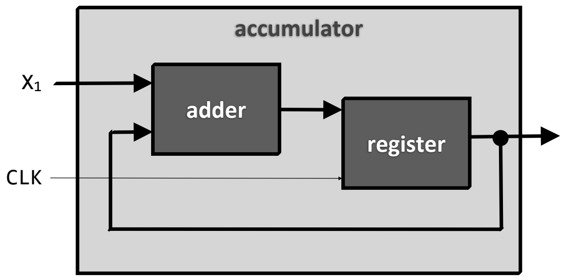


Figure 1.9: Accumulator structure. The accumulator can be constructed by combining a register and an adder.

1.2.4 Implementing systems in software

Systems can be implemented and simulated in software using a programming language. The notion of stateless system perfectly matches the concept of a function in a structured programming language. System inputs are mapped to a function input argument while system outputs are mapped to the function return values.

For example, the adder can be easily implemented as a MATLAB® function:

```
function Y=adder(X1,X2)
% Returns the sum of X1 and X2
Y=X1+X2;
end

>> z=adder(3,2)
z =
     5
```

This a time-discrete implementation since the function is invoked at some given time instants.

System composition can be easily achieved by making use of the return value of a function as the input argument of another function. As an example consider implementing the 3-operand adder using a 2-operand adder:

```
function Y=adder3(X1,X2,X3)
% Returns the sum of X1, X2 and X3
temp=adder(X1,X2);
Y=adder(temp,X3);
end

z=adder3(2,3,5)
z =
    10
```

Sateful systems offer some more difficulty, since we need to implement the memory property. This can be implemented in MATLAB® using “persistent” variables (other languages would require “global” variables). This variables hold their value across invocations.

For example, the accumulator example could be implemented in MATLAB® as follows:

```
function Y=accumulator(X)
% Returns the previous output value plus X
persistent reg;
if isempty(reg)
    reg=0;
end
reg=adder(X,reg);
Y=reg;
end

>> z=accumulator(1)
z =
     1
>> z=accumulator(3)
z =
     4
```

The use of persistent variables does not meet the concept of “pure function” in programming. Pure functions are stateless: outputs are a function of current inputs exclusively. However, persistent variables are useful to illustrate the memory property and to implement stateful systems. On the other hand, stateless systems perfectly match the concept of pure function.

1.2.5 Finite-state machines

A *finite-state machine* is an automaton whose behavior can be described by a finite number of states (Kohavi and Jha 2010). The machine can be in exactly one of a finite number of states at any given time. An example of a finite-state machine could be the *vending machine* of figure 1.3. The vending machine behavior can be described by four possible states:

- **Idle**: waiting for some product to be selected.
- **Waiting coins**: waiting for the product to be paid.
- **Serving product**: delivering the product to the customer.
- **Returning coin**: returning the coin if the product has been cancelled.

Initially the machine is in idle state waiting for some product to be selected. Selecting a product activates the machine that shows the product price and then evolves to an state where it waits a coin to be deposited in a slot or to cancel the selection. Putting a coin makes the machine to serve the product and going back to the idle state after some time. Canceling the selection makes the machine to return the coins and return to the idle state.

The machine can change from one state to another in response to some external inputs; the change from one state to another is called a *transition*. The events that can trigger transitions in a vending machine are:

- *push*: selecting some product on the machine keyboard.
- *coin*: putting a coin in the slot to pay the product.
- *cancel*: cancelling the selection.
- *time*: event that denotes that 3 seconds elapsed from the last transition.

A finite-state machine is defined by the list of its states, its initial state, and the transitions between states. The finite-state is usually defined by a *state diagram* or a *state transition table*. A state diagram is a graph where states are represented as nodes and transitions are represented as arrows between states. Figure 1.10 shows the state diagram of the vending machine.

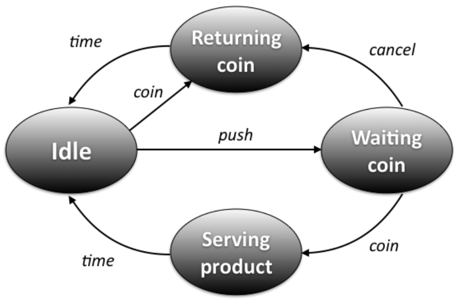
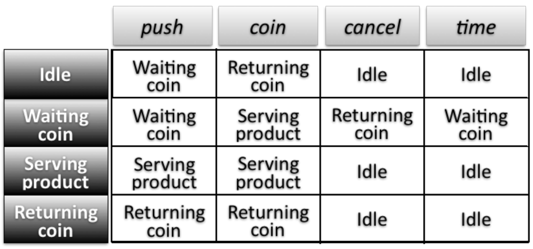


Figure 1.10: State diagram of the vending machine. Nodes represent states and arrows represent transitions.

A state transition table is a table that specifies the next state of the machine upon the occurrence of a transition. Figure 1.11 shows the transition table of the vending machine. It is equivalent to the state diagram of figure 1.10, but it specifies thoroughly all transitions.

Note that a finite-state machine is a stateful machine since the next state is always a function of the previous state and the event. The current state needs to be memorized in order to properly respond to a new event.



	push	*coin*	*cancel*	*time*
Idle	Waiting coin	Returning coin	Idle	Idle
Waiting coin	Waiting coin	Serving product	Returning coin	Waiting coin
Serving product	Serving product	Serving product	Idle	Idle
Returning coin	Returning coin	Returning coin	Idle	Idle

Figure 1.11: Transition table of the vending machine. It specifies the next state after the occurrence of a transition.

Finite state machines can be better understood through a program that simulates them. The program of figure 1.12 provides an implementation of the skeleton of a vending machine as a finite-state automaton. If you are not still familiar with programming, do not worry, just run the program and see how it works to get a flavor of what an state automaton is and skip the rest of this paragraph. The program implementation can be introduced as a programming exercise and properly understood when you are already familiar with programming basics. But if you are keen to know how it has been implemented, the main ideas are outlined next.

The system state needs to be a persistent variable. The key of this implementation is table `tr_table` in line 11. It provides the new state as a function of the previous state and the event. Inputs like *push*, *coin* or *cancel* need to be specified as a user input. The *time* event is handled is a different way: it is produced automatically after a pause of 3 seconds.

A sample execution of the vending machine program is shown in figure 1.13. Lines 1-12 show a normal cycle of the vending machine. Notice how “non-expected” inputs, like the one of line 15, or invalid events, like the one of line 23, are handled.

Vending machines are nowadays implemented using a small computer inside. But think in one of those antique and beautiful mechanical implementations of the 50s and the 60s. The implementation was completely different, but their behavior was yet the same of the program of figure 1.12. The vending machine is a nice example of an automaton. Although beautiful, the internal machinery of old vending machines was so specific for this application that it could not be reused to solve any other task. With the years, computer based implementations would replace these old implementations. The next section explains why a computer is strictly more powerful than a finite-state automaton.

```
function vending_machine
clc; clear all;
persistent state;
if isempty(state)
    state=1; % Idle state
    next_state=1;
end

state_names={'Idle', 'Waiting coin', 'Serving product', 'Returning
    coin'};
event_names={'push', 'coin', 'cancel', 'time'};
tr_table=[2,4,1,1;
    2,3,4,2;
    3,3,1,1;
    4,4,1,1];

while(true)
    % In states 1 and 2 some keyboard typing is needed
    if (state==1 || state==2)
        event=input('1.- push, 2.- coin, 3.-cancel\nSelect an event:
            ');
        if (event<1 || event>3)
            fprintf(2,'Invalid event\n');
            next_state=state;
        else
            fprintf('Event: %s\n',event_names{event});
            next_state=tr_table(state,event);
        end
    % States 3 and 4 are time triggered
    else
        event=4;
        fprintf('Event: %s\n',event_names{event});
        pause(3);
        next_state=tr_table(state,event);
    end

    state=next_state;
    fprintf('State: %s\n\n',state_names{state});
end
end
```

Figure 1.12: Matlab program of the vending machine. Table `tr_table` provides the new state as a function of the previous state and the event. The state needs to be a persistent variable.

Para seguir leyendo haga click aquí