Design of digital systems

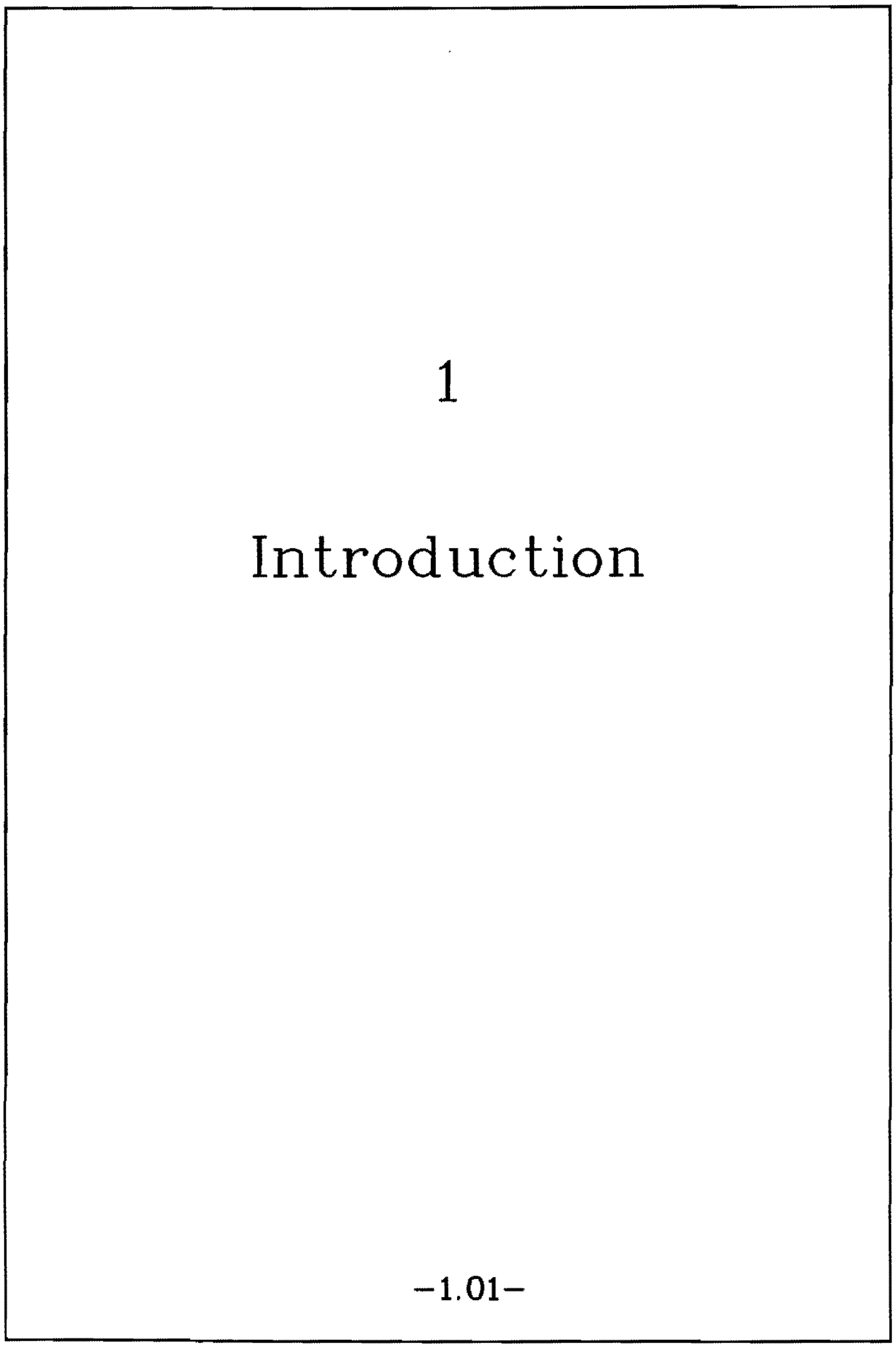
Eindhoven International Institute

Course In Electronic Engineering

Survey	Course:
Nr 276/2	: **INTRODUCTORY SEMESTER 1993** **- Electronic Engineering -**
Subject	: DESIGN OF DIGITAL SYSTEMS
Lecturer	:Ir M.J.M. van WEERT
Author	: Ir M.J.M. van WEERT
Copy	:

Problems: Problem numbers in parentheses refer to the problem numbers in the Dutch edition of this survey (edition January 1992)

Table of Contents



1

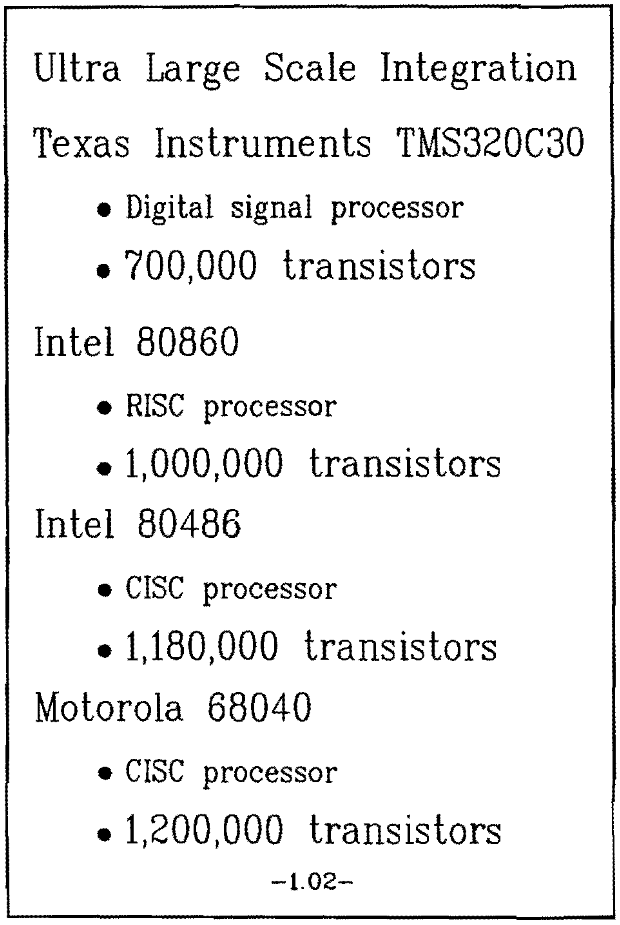
Introduction

−1.01−

1 Introduction

This course deals with the design of digital systems, being systems having discrete rather than continuous inputs and outputs.

The central -but certainly not trivial - question in this course is how to design such systems.



Ultra Large Scale Integration

Texas Instruments TMS320C30

- Digital signal processor
- 700,000 transistors

Intel 80860

- RISC processor
- 1,000,000 transistors

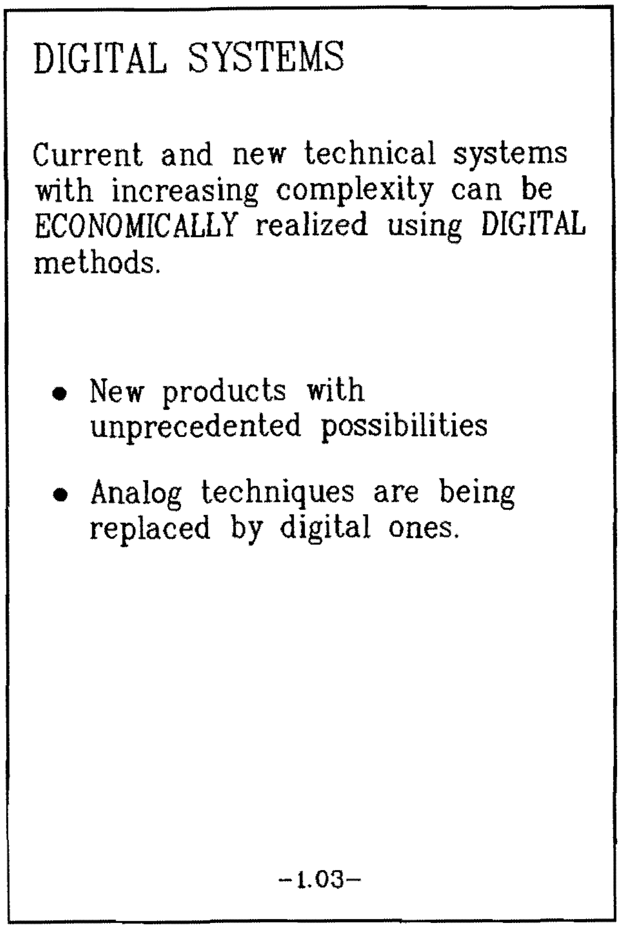
Intel 80486

- CISC processor
- 1,180,000 transistors

Motorola 68040

- CISC processor
- 1,200,000 transistors

–1.02–



DIGITAL SYSTEMS

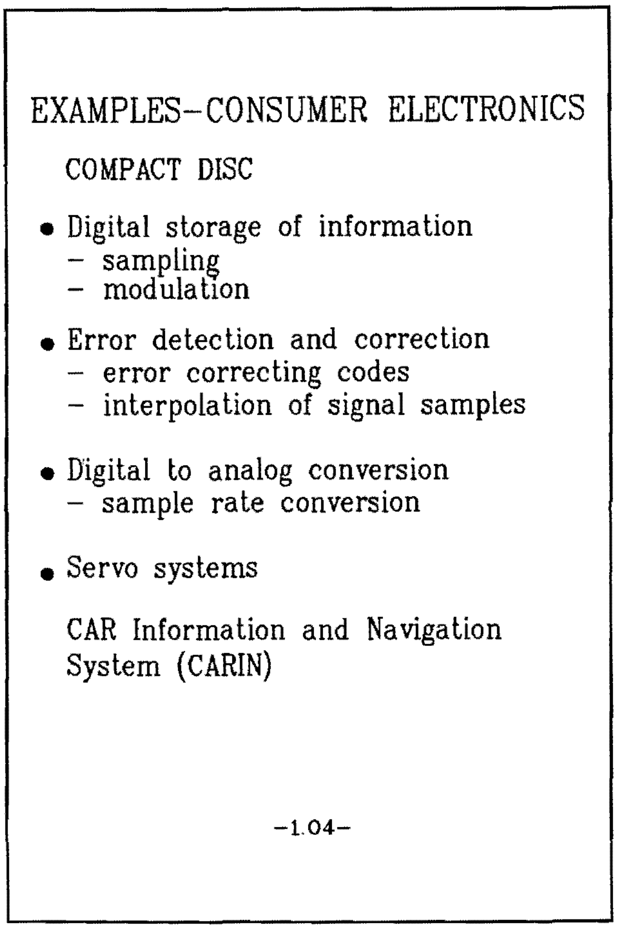
Current and new technical systems with increasing complexity can be ECONOMICALLY realized using DIGITAL methods.

- New products with unprecedented possibilities
- Analog techniques are being replaced by digital ones.

–1.03–

As a result of the progress of IC-technology we are able to build continually larger and more complex, but still economic, digital systems. Figure 1.02 shows an example of the state of the art in 1990. We see four examples of a pre-eminent digital system: the microprocessor. The first one is a digital signal processor with 700,000 transistors, a RISC processor with 1,000,000 transistors (RISC = Reduced Instruction Set Computer). The second one is a Complex Instruction Set Computer, a so-called CISC-processor, with 1,180,000 transistors and another CISC-processor with 1,200,000 transistors. Figure 1.02 lets us see that the complexity of a digital system, expressed in the number of transistors on a single IC, has exceeded the 1,000,000 transistor boundary. We have entered the era of Ultra Large Scale Integration (ULSI). Furthermore, we do not expect development to stop here, and time should bring a continually growing complexity.

Up till now we have considered digital systems on a single IC. The complexity of digital systems realized on one or more printed circuit boards, with the help of more integrated circuits, will exceed multiples of this complexity.
This course deals with the state of the art of the design of such gigantic systems.
The central question remains: "How do we design such a complex digital system?". However, before we answer this question, or point to possible directions, we will first take a look at the importance of digital systems.
The importance of these systems can not be separated from the powerful technological development that is due to the production of integrated circuits. Actually, the high standard of the microelectronics technology is what enables us, the digital system builders, to economically realize complex digital systems. This means, as figure 1.03 shows, that for existing or new technical problems we can still realize economically feasible (i.e. affordable), complex digital solutions. This will lead to new products with unlimited possibilities. This will also lead to systems where previous analog solutions are replaced by digital ones. Digital systems, hand in hand with microelectronics, are becoming increasingly important in all sorts of subjects and are infiltrating more fields. It is difficult not to fall in the temptation of filling a whole chapter with all sorts of criteria without thinking of an example. However, we shall limit ourselves to discussing examples in the field of audio-visual applications; the field of consumer electronics.



EXAMPLES-CONSUMER ELECTRONICS

COMPACT DISC

- Digital storage of information
  - sampling
  - modulation
- Error detection and correction
  - error correcting codes
  - interpolation of signal samples
- Digital to analog conversion
  - sample rate conversion
- Servo systems

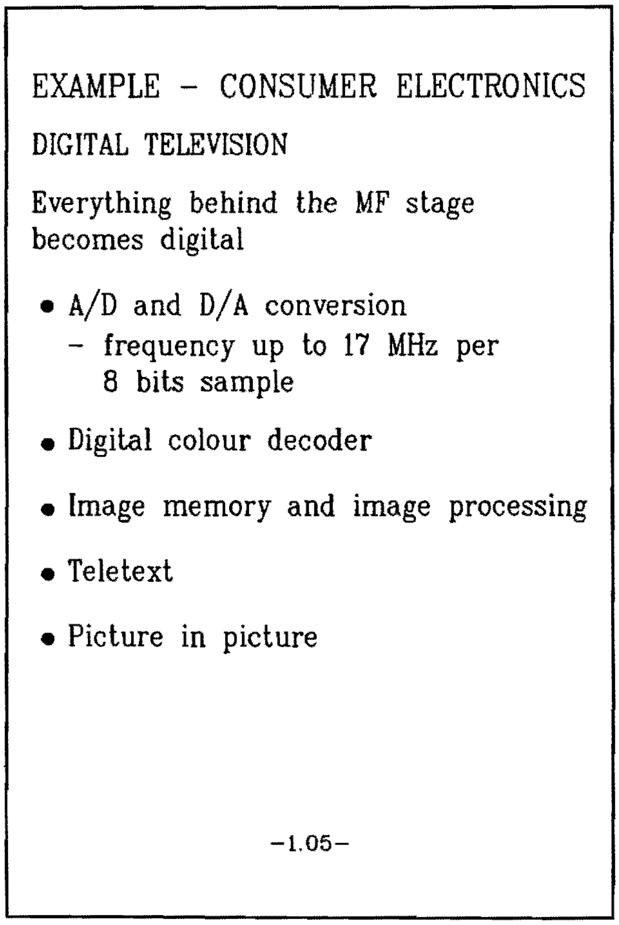
CAR Information and Navigation System (CARIN)

-1.04-

An example in the consumer electronics area is the compact disc, see figure 1.04. In the compact disc system, audio signals, music and speech are digitally written on a medium, the compact disc. They are digitally read back with the help of light. Digital storage means that audio signals being analog by nature, have to be translated to digital signals. The sampling and quantizing of the signal does not belong to the field of this course. Some aspects of coding will be discussed later. Once we have the signal in digital form we can do with it all sorts of things using digital systems. We can process the signal; examples being filtering and modulation. Because the signal is digital we can always reconstruct the original signal.

When writing or reading information to/from a compact disc something may go wrong, errors can occur. A solution is the use of error detection/correction codes and the interpolation of signal samples. This is also an example of the application of advanced techniques in a digital system. The servo system in a compact disc is also very important, i.e. the control system of the laser head and motors. There are also many digital techniques applied there.

Besides the compact disc, there are other derived products, such as the CD-ROM, and the Car Information and Navigation System (CARIN), where digital systems play an important role.



EXAMPLE - CONSUMER ELECTRONICS

DIGITAL TELEVISION

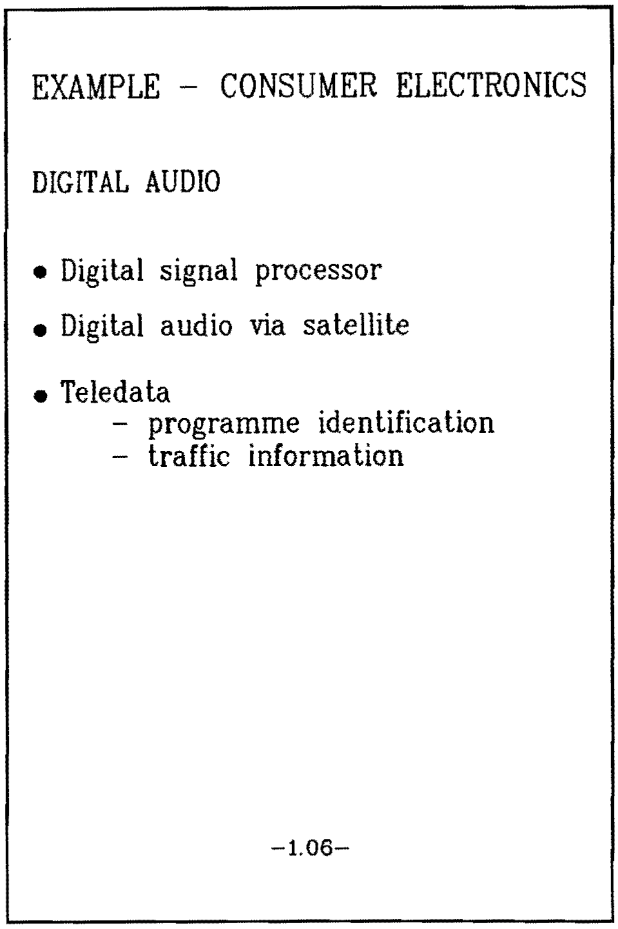
Everything behind the MF stage
becomes digital

- A/D and D/A conversion
  - frequency up to 17 MHz per 8 bits sample
- Digital colour decoder
- Image memory and image processing
- Teletext
- Picture in picture

-1.05-

Another example in the field of consumer electronics is the digital television, figure 1.05. The expectation is that base band audio and video signals will be processed digitally. Video signals occupy a larger frequency range than audio signals. This is expressed in higher sampling frequencies in digital television, up to 17 MHz. In the processing of digital video signals we can first consider items such as digital colour decoders. This will lead to a better quality of the decoding process with lower costs. Also, we can consider saving an image in memory and processing it later on: image processing. The plans for high definition TV can only be realized by reducing the required bandwidth through the use of image processing techniques.

Other examples of the application of digital techniques in consumer-TV are Teletext and "picture in picture".



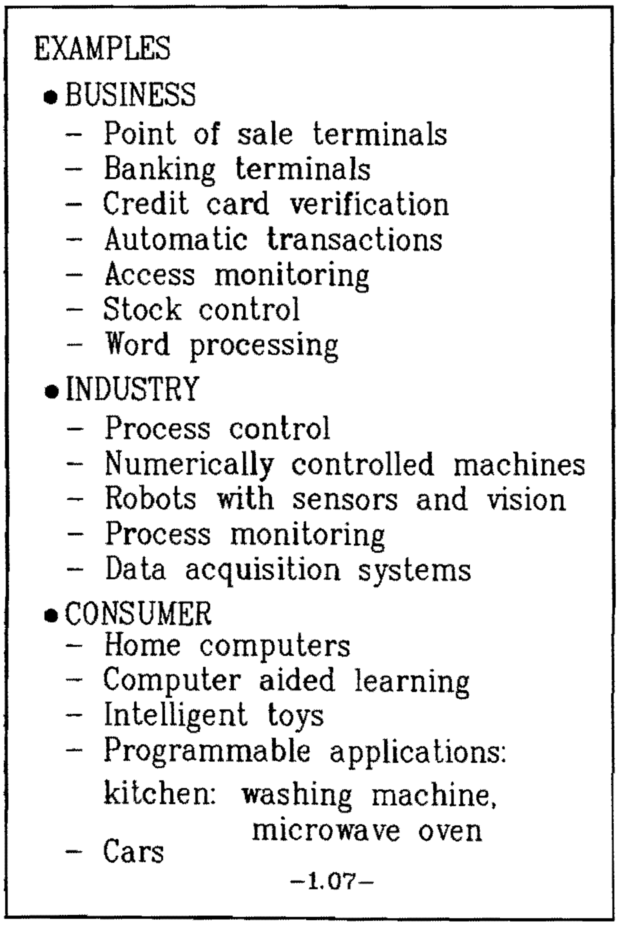
EXAMPLE - CONSUMER ELECTRONICS

DIGITAL AUDIO

- Digital signal processor
- Digital audio via satellite
- Teledata
  - programme identification
  - traffic information

-1.06-

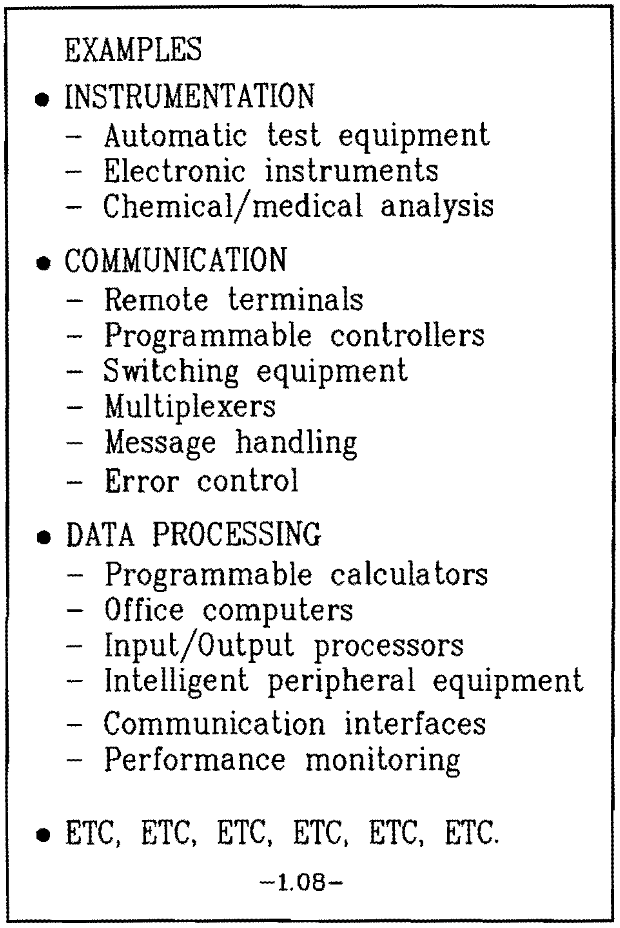
As a third example in the field of consumer electronics we mention digital audio. In figure 1.06 we see that digital signal processors are currently in a state that allows them to be used in all sorts of audio signal applications. These digital system processors could be more general, or specifically designed for a special purpose application. We have arrived at a state where a digital system processor can replace the classical sound controls of an amplifier. Another application of digital audio is the transmitting of radio programs, digitally, via a satellite. It is also possible to transmit other information in addition to the program information. We are then talking of tele-data. Examples of applications could be program identification, as well as traffic information. For example, information can be distributed via such a system in the from of files. This information can then be used in a system, such as the previous mentioned CARIN system, to point out alternative routes. These are examples of applications where digital systems play an important role and that are currently technically implementable.



EXAMPLES

- BUSINESS
  - Point of sale terminals
  - Banking terminals
  - Credit card verification
  - Automatic transactions
  - Access monitoring
  - Stock control
  - Word processing
- INDUSTRY
  - Process control
  - Numerically controlled machines
  - Robots with sensors and vision
  - Process monitoring
  - Data acquisition systems
- CONSUMER
  - Home computers
  - Computer aided learning
  - Intelligent toys
  - Programmable applications:
    kitchen: washing machine,
    microwave oven
  - Cars

-1.07-



EXAMPLES

- INSTRUMENTATION
  - Automatic test equipment
  - Electronic instruments
  - Chemical/medical analysis
- COMMUNICATION
  - Remote terminals
  - Programmable controllers
  - Switching equipment
  - Multiplexers
  - Message handling
  - Error control
- DATA PROCESSING
  - Programmable calculators
  - Office computers
  - Input/Output processors
  - Intelligent peripheral equipment
  - Communication interfaces
  - Performance monitoring
- ETC, ETC, ETC, ETC, ETC, ETC.

-1.08-

In addition to consumer electronics, there are many other fields where digital systems are applied. Figures 1.07 and 1.08 mention a number of them. We see first examples from the commercial field, such as cashier terminals, banking terminals, identification of credit cards, automatic banking transactions etc., collectively known as point of sales systems. We also see applications in building security, stock control, word-processing etc. As examples of industrial applications we can think of things such as process control, numerically controlled machines, robots with sensory and vision capabilities, automatic assembly lines etc. In general: process control and data acquisition systems.

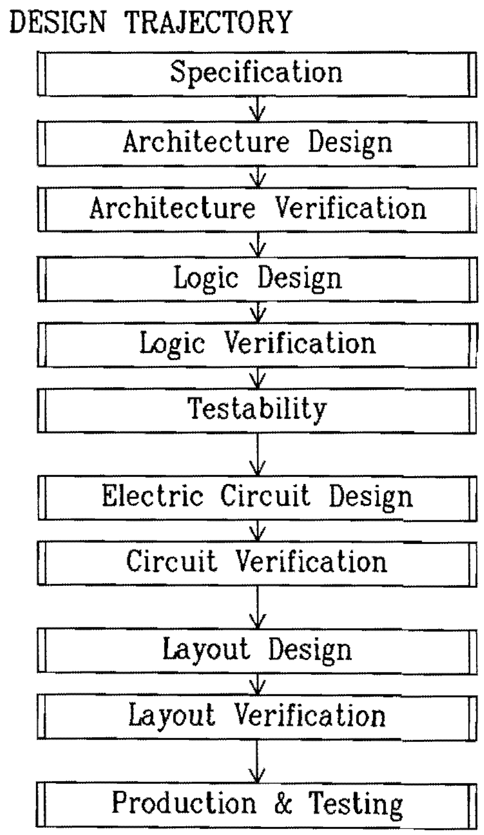
As examples of other consumer applications one could name home computers, game computers, tutoring systems, intelligent toys, programmable home appliances such as washing machines, microwave ovens etc. Also applications in automobiles such as the ignition and break systems. From the field of instrumentation we think of testing devices where many functions could be automated, for example continuous automatic calibration. By instrumentation we mean electronic as well as chemical and medical analysis instruments.

One of the roots of digital techniques lies in the field of telecommunication. There we find yet another important application field of digital systems. For example, remote terminals, programmable controllers, switching devices for telephones, multiplexers for data transmission etc. A second important application area of digital techniques is the field of processing and storage of data and information, i.e. the field of data processing. Such applications include programmable calculators, office computers, I/O processors, intelligent terminals and large computers connected to local area networks.

In this overview we did not pursue completeness; there are still many unmentioned fields.

Thus far we have discussed a large number of application fields of digital systems, having a complexity of millions of transistors. This complexity will continually increase in the future.

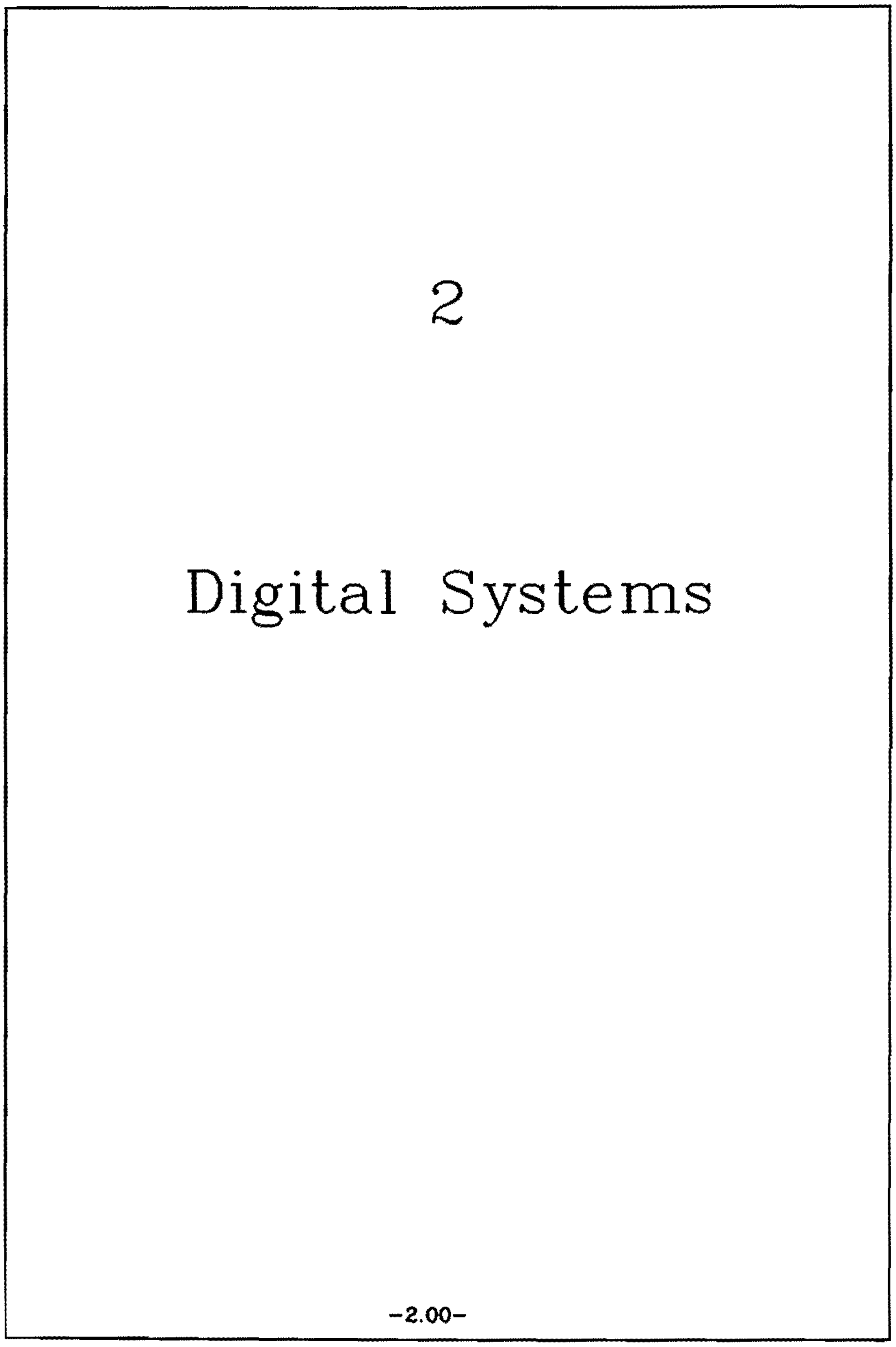
We now return to the central question of this course: "How do I design a digital system with such a complexity?". It will be clear that designing large systems by starting with a network of a few transistors, and then expanding it until the total system is realized, will not yield acceptable results.



-1.09-

This is a methodology that would probably lead to disappointment. A better strategy is shown in figure 1.09.

Digital design is a structured and planned occupation according to a checklist. The whole design comprises a number of phases, beginning with an idea and ending with the production and testing of the final system. We begin by specifying what we want to build. Next follows the so-called *architecture* phase. In this phase we consider how the digital system should behave in order to satisfy our requirements. The activity of whether the chosen architecture satisfies our specifications is called *verification*. After the architecture phase we start the *logic design* phase. In this phase the building blocks of the architecture are translated into realizable logic circuits. We utilize a computer simulation system to check that the logic circuitry is indeed an implementation of the specified architecture. We are then speaking of *logical verification.* An important aspect of the design is *testability*. How do we ensure that an IC with 1,000,000 transistors functions correctly after production?. During the logical design we have to take measures that will guarantee this testability. That is to say making testing simpler. At the end of the logical design phase we have a number of schemes with logical building bricks. These schemes must be translated into schemes with transistors, resistors, diodes, and similar components. This translation process we call the *electronic circuit design*, being the next phase in our design trajectory. In this phase we start a new specification. After the transistor schemes have been designed we start the *layout* phase in which we make the IC layout. That is, we show where the transistors should be placed on the IC, how they are connected etc. After the layout verification follows the production and the testing. Figure 1.09 suggests that all these activities in the design process, are carried out by one person. In reality this is generally not true. The specification is made by the customer, mostly in cooperation with the system designer. The architecture and logic design are generally made by the digital system designer. The electrical circuit design and the detailed layout is the domain of the IC designer. The digital system designer will remain globally involved in this phase. The testing of the finished product is carried out by the test engineer. In this course we will be occupied with the design of digital systems. That is to say: we are considering methods for systems specifications. We will direct out attention towards architecture. Another important topic in this course is the methodology and techniques used in logic design.



2

Digital Systems

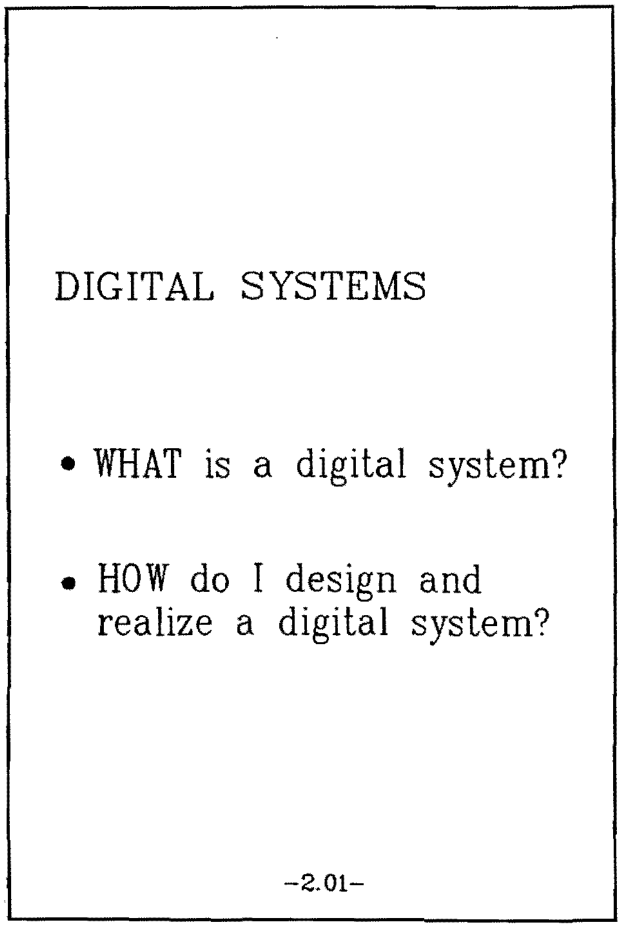
-2.00-

2 Digital Systems

2.1 What is a digital system?

As previously mentioned this course covers the design and realization of digital systems.

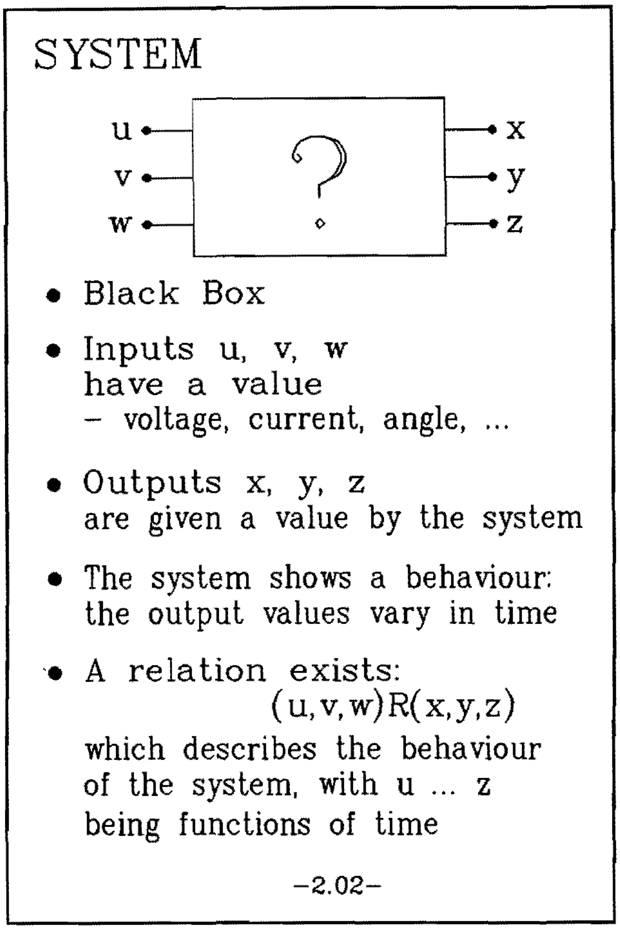
This should be seen in the context of the design trajectory mentioned in the previous chapter.



DIGITAL SYSTEMS

- WHAT is a digital system?
- HOW do I design and realize a digital system?

-2.01-



SYSTEM

- Black Box
- Inputs u, v, w have a value
  - voltage, current, angle, ...
- Outputs x, y, z are given a value by the system
- The system shows a behaviour: the output values vary in time
- A relation exists:
  $(u,v,w)R(x,y,z)$
  which describes the behaviour of the system, with u ... z being functions of time

-2.02-

The topic of design presents two questions (see figure 2.01):

- What is a digital system?

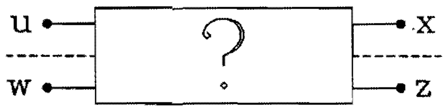
- How do I design and realize a digital system?

Before we discuss the design and realization of a digital system, we must first know what a digital system is. What makes something a digital system? Fortunately we have notion as to what that is. If I ask you what a digital system is, you will frequently come with answers such as: a computer, (parts of) an automaton, such as a coffee machine, traffic light controllers, systems in consumer electronics, etc. And also clocks, such as those hanging all over the university, are examples of digital systems. We shall limit ourselves, in this course, to digital systems that are built using electronic building bricks.

The question: "What is a digital system?", comprises two sub-questions. First: "What is a system?", and "What are the special characteristics of a digital system?". Formally speaking we can think of a system as an object, or a black box, with inputs and outputs (see figure 2.02). We shall limit ourselves, in this course, to systems with a finite number of inputs and outputs. The inputs (here called u, v, and w) have a value. This could be a voltage, a current, an angle etc. The outputs (called here x, y, and z) are assigned a value by the system. This could also be a voltage, a current etc. A system displays its behaviour. That is to say the output values change in time due to the change of one or more of its inputs. The outputs of the systems we consider are not arbitrary. They are systematic, rule governed, and the outputs change their values in time according to a deterministic behaviour. We could say that the temporal behaviour (behaviour with respect to time) is described by the system as a relation between the input and output values.

DIGITAL SYSTEM

- Each input or output has a value range (domain) with a <u>finite number</u> of different values



$u \in I_u\{\ \ldots\ldots\ \}$ finite set

$w \in I_w\{\ \ldots\ldots\ \}$ finite set

$x \in O_x\{\ \ldots\ldots\ \}$ finite set

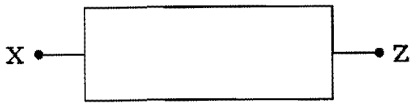
$z \in O_z\{\ \ldots\ldots\ \}$ finite set

A relation exists

$(u,..,w)R(x,..,z)$

-2.03-

EXAMPLE - SYSTEM



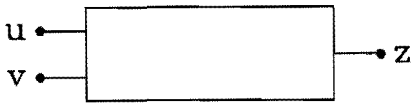
$x,z \in \{$real numbers between 0 and 1$\}$

relation: $z=F(x)=\sqrt{x}$

THIS IS <u>NO</u> DIGITAL SYSTEM

Value range of input and output comprises an infinite number of elements

EXAMPLE - SYSTEM



$I_u, I_v = \{\ 0\ ..\ 9\ \}$

$O_z\ = \{$ yes , no $\}$

$$F : z = \begin{cases} \text{yes if } u > v \\ \text{no otherwise} \end{cases}$$

THIS IS A DIGITAL SYSTEM

-2.04-

A digital system (figure 2.03) is a system with the (extra) property that all inputs and outputs have a value domain with a finite number of values. The value domains of u, v, and w form a finite set, i.e. a set with a finite number of elements. Furthermore, the values of the outputs x, y, and z form a set with a finite number of elements. Moreover, it still holds, for digital systems, that there is a relation between inputs and outputs.

This relation, R, specifies the behaviour of the system in time, and shows the response of the system to the change in input values. We observe, however, that a specific set of input values does not necessarily give a fixed response. Even the *system itself* does not change in time. The system remains the same but it can have memory (it can remember), involving the existence of system *states*. In this case, we are considering a *sequential system*. We shall return to this topic later in this course. We remind ourselves that we shall limit our discussion to systems that do not change their behaviour in time, the so-called *time-invariant systems*.

In figure 2.04 we have two examples of what could be, and what is not a digital system. The first example shows a relation, or a function where the input and output values belong to the set of real values between 0 and 1, and the relation is given by

$$z=\sqrt{x}$$

This is not a digital system because the value domains of input and output do not contain a finite number of values. In our second example we have a system with two inputs, u and v, where the values of u and v belong to the set of whole numbers from 0 to 9. We are going to use, throughout this course, a Pascal-like notation to denote such sets. Furthermore, the system has an output z, with a value domain of "yes" and "no". The relation is given by a function, F, defined by z = "yes" if u > w and "no" otherwise. This is an example of a digital system because input and output values form finite sets. Clocks, such as those hanging all over the university, are examples of digital systems. The input is composed of a minute pulse or, more accurately, its presence or absence. The set of output values contains all the possible positions of the minute and hour hands: sixty positions for the minutes hand and twelve for the hours hand. Thus finite sets in both cases.

BLACK BOX MODEL

DIGITAL SYSTEM

I R O

Input – Relation – Output

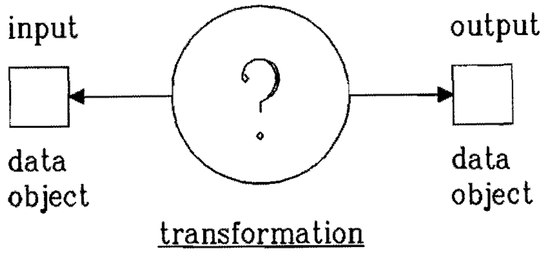
- R: Specifies the behaviour of the system
- R: Changes in inputs values

  →

  Changes in output values
- R: Is a specification

-2.05-

ALGORITHMIC MODEL



- transformation rules describe conversion of data object from input domain to output domain
- a number of more simple transformations

  →

  prescription, "algorithm"
- system behaviour specified by algorithmic description, "programme"

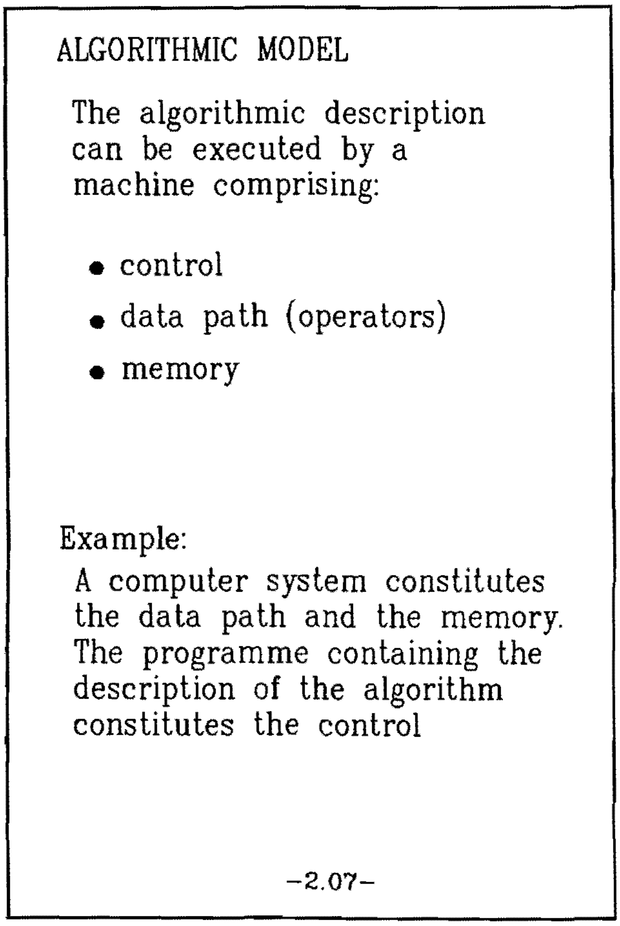
-2.06-

To summarize (see figure 2.05) a digital system is one with several inputs and outputs where all inputs and outputs have their own domain. Furthermore, these value domains have a finite number of elements. In addition, there is a relation, R, between inputs and outputs, i.e. between input domains I and output ranges O. This relation, R, defines and specifies the behaviour of the system. Thus R defines how the system changes its output values, when the input values change. We also call R a specification of the system. This relational view of the system behaviour and system specification is called the *black box model*, or sometimes the *system model*.

If we look at a digital system from a different perspective, we can, rightfully, say that it samples its inputs before generating the corresponding outputs (see figure 2.06).

The system processes the input values to produce new output values, i.e. transforms input values (from the set of possible input values) to output values (from the set of possible output values). Thus, the digital system performs a transformation of data objects from the input domain to data objects of the output domain. The rules defining this transformation accurately describe the conversion of objects from the input domain to objects from the output domain. These transformation rules specify the behaviour of the system. In general the transformations, executed by digital systems, are very complex. We shall have to decompose such transformations into simpler ones. These simple transformations may be (partly) carried out in parallel or in series, so that the same overall result is obtained.

We receive a recipe or prescription for the production of the output data from the input data. We shall call such a prescription an *algorithm*. In this view towards the system, the behaviour is defined by a formal algorithmic description, a program. Now we shall consider the algorithmic model. We can regard such an algorithmic description as a specification of the system and also as a realizable description.



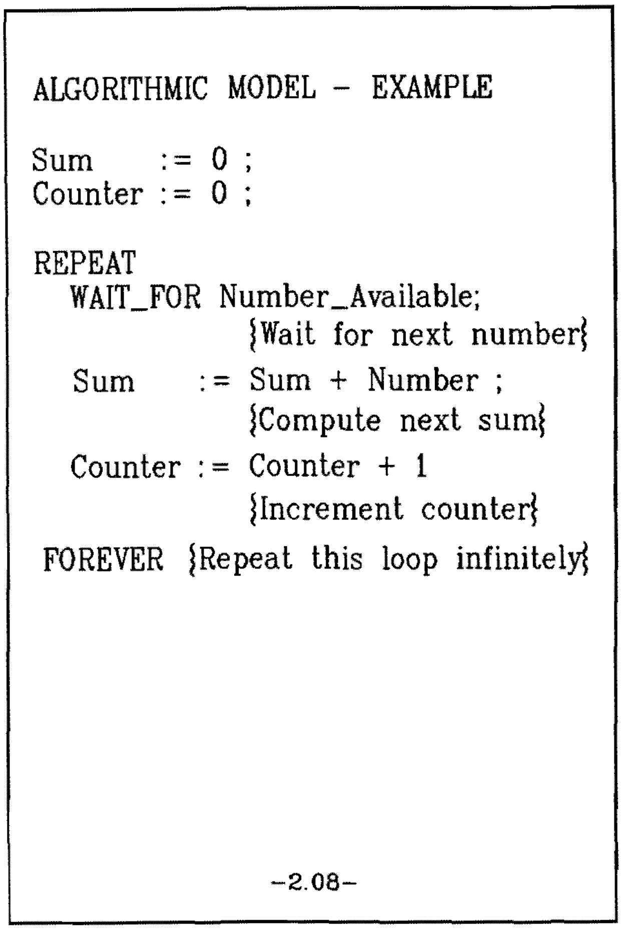
ALGORITHMIC MODEL

The algorithmic description can be executed by a machine comprising:

- control
- data path (operators)
- memory

Example:
A computer system constitutes the data path and the memory. The programme containing the description of the algorithm constitutes the control

-2.07-



ALGORITHMIC MODEL - EXAMPLE

```
Sum     := 0 ;
Counter := 0 ;

REPEAT
  WAIT_FOR Number_Available;
                {Wait for next number}
  Sum     := Sum + Number ;
                {Compute next sum}
  Counter := Counter + 1
                {Increment counter}
 FOREVER {Repeat this loop infinitely}
```

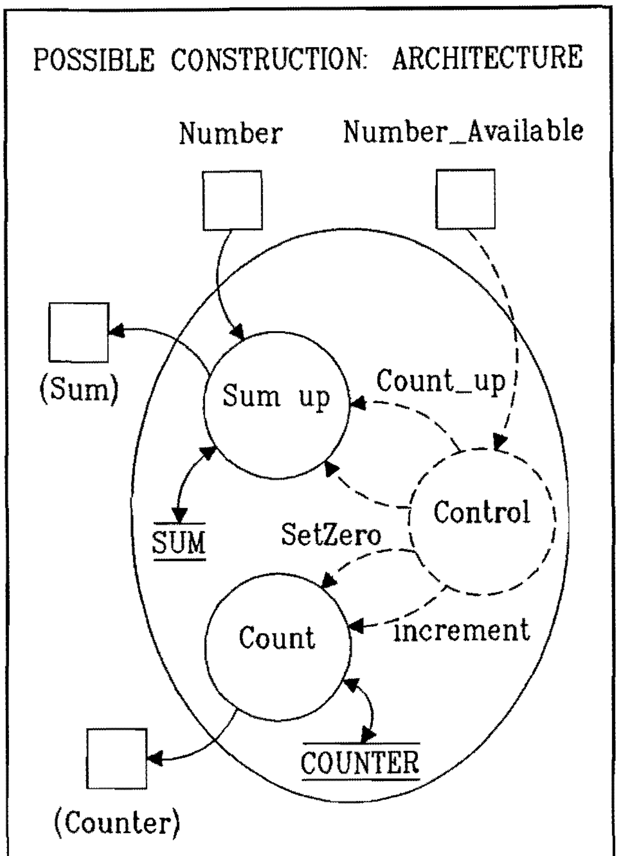
-2.08-

An algorithmic description is, indeed, executable by a machine comprising the following elements (see figure 2.07):

- Management or control
- Operators or data path
- Memory or data storage

The control defines when each (partial) transformation may take place. The data path or operators carry out the transformation. The memory or data storage serves for the preservation of the intermediate results and inputs.
Consider as an example a computer system running a program. The computer system forms, from our viewpoint, the data path and the memory of the system. The program which describes the algorithm, forms the control. Thus, following this model, we can easily make a system that satisfies the specification. A disadvantage, however, is that such a realization will be slow and expensive in comparison with tailored realizations.
Let us look at an example of an algorithmic system description. Consider a system that adds a list of successive numbers and displays, as a result, the running sum and the count of added numbers. A description of the system's behaviour is given in figure 2.08. We have used a Pascal-like language to describe the system, we shall do so throughout this course. Achieving an absolute fluency in a programming language, such as Pascal, is not the purpose of this course. However, we shall use a Pascal-like language for the communication of behavioral descriptions. For this purpose, we do not need too much language knowledge and fluency.

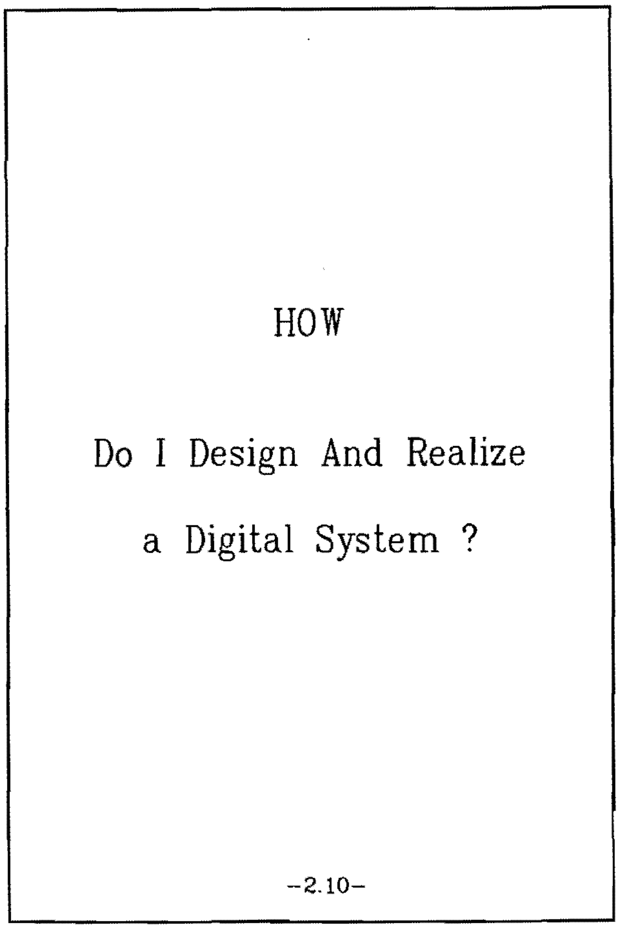
We can notice the following in the behavioral description of figure 2.08. The system begins by initializing its memory. That is to say, Sum and Counter are set to 0. The next construct is a "REPEAT FOREVER" construct. Throughout this course we shall use similar constructs to denote that the system behaviour changes in time, hence that the system has memory. The system will respond differently to the same input. For example, the same input, e.g. 5, will produce the successive Sum output: 0, 5, 10, 15, etc. Furthermore, we see from the system description that it (obviously) waits until the following input number is ready. Next, the number is added to the running sum and the counter is incremented by 1. This is the whole system behaviour.



-2.09-

Figure 2.09 shows a sketch of a possible architecture. Here we make use of the three previously mentioned elements: control, operators and memory. The control is denoted by a dashed circle, the operators by a solid circle and the memory by two horizontal lines. Examples are the memory of the sum and counter. We observe that the behavioral description introduces two transformations, which leads to equivalent operators or data paths in our model, namely the summation operator and the count operator. The new running sum is computed by the summation operator. The count operator increments the counter by one when activated. The total system in managed by a controller. When the controller receives an external command, indicating that a new number is available, it will signal the sum and count operators to start work. Later in this course we shall go deeper into the use of these models.

We have seen a short global answer of the question of "What is a digital system?". A digital system's input and output elements belong to finite sets of possible values. Furthermore, in this course we shall limit ourselves to digital systems that can be realized with electronic components and that are time invariant. The behaviour of such a system can be described by the relation between input and output values. In this case we are considering a black box model. Moreover, this behaviour could be described by a set of transformation rules or algorithms. In this case we are considering an algorithmic model. In the latter case, the realization in a data-path/control model is possible, where each part could be described by a black box model.



HOW

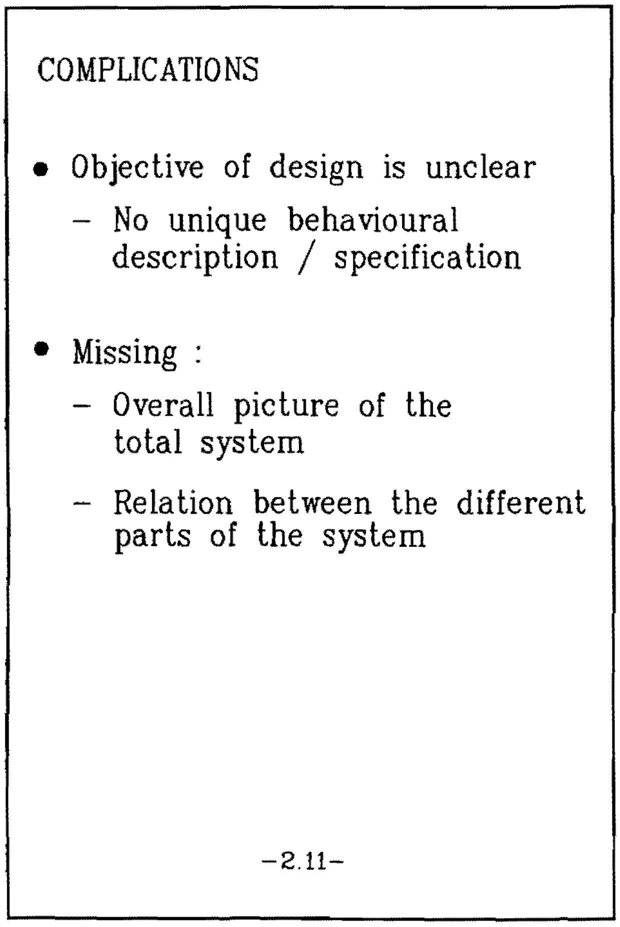
Do I Design And Realize

a Digital System ?

–2.10–

2.2 The design of a digital system

After having discussed the question "What is a digital system?" we have arrived at the central topic of this course: "How do I design a digital system?". The ultimate purpose of a design process is the realization of a digital system built of easily available, trustworthy, standard components such as transistors, resistors, logic gates, flip-flops, counters, registers, and also more complex standard building blocks such as microprocessors, I/O ports, etc. The system that we build from these standard components should behave in a predefined way and exhibit this predefined behaviour. In practice this is not as simple as described here.

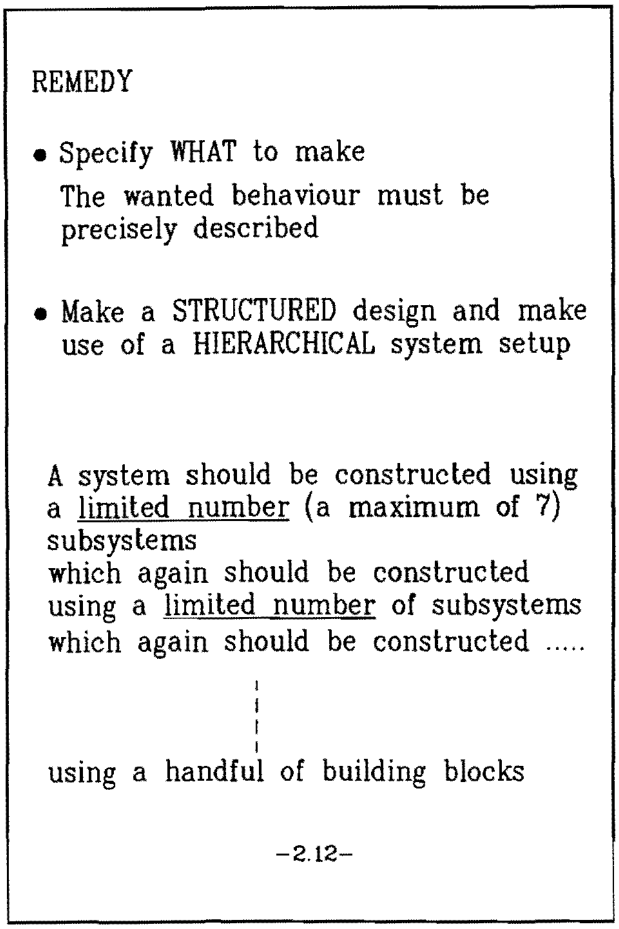


COMPLICATIONS

- Objective of design is unclear
  - No unique behavioural description / specification
- Missing :
  - Overall picture of the total system
  - Relation between the different parts of the system

-2.11-

Two frequently occurring complications are (see figure 2.11): the absence of a (good) behavioral description. That is, what the system must do is not (clearly) specified. This results in a situation where the realized system does not always do what the client had in mind. The swing example on page 2.9 illustrates this problem. Secondly, when dealing with very complex systems one tends easily to lose the overview of the total system. One has frequently no insight in the relation between different system subcomponents. This results in situation where suboptimal solutions are frequently obtained, sometimes causing sub-components not to work optimally together. When the underlying relation is not clear anymore, one often attempts to solve a problem in one part which actually occurs in other parts.



REMEDY

- Specify WHAT to make

  The wanted behaviour must be precisely described

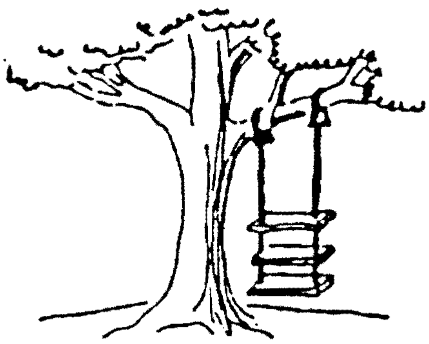
- Make a STRUCTURED design and make use of a HIERARCHICAL system setup

A system should be constructed using
a <u>limited number</u> (a maximum of 7)
subsystems
which again should be constructed
using a <u>limited number</u> of subsystems
which again should be constructed .....

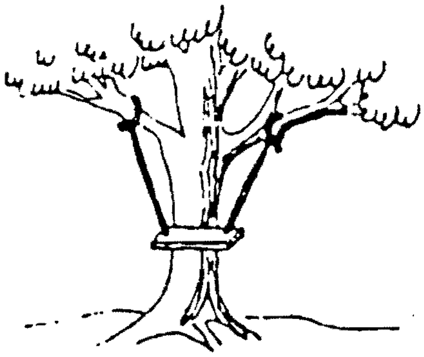
using a handful of building blocks

-2.12-

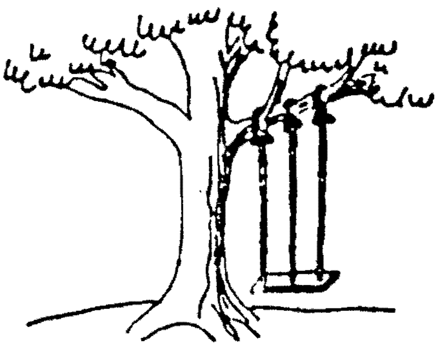
The remedy of these design complications is twofold (see figure 2.12). First we have to decide WHAT we shall make. That is, the desired system behaviour must be carefully described. Secondly, we must manage the complexity by designing in a structured way. Therefore we must use a hierarchical system structure. This means that we shall try to build a system from a limited number of subsystems (a maximum of 7). We shall try to construct each of these subsystems from a limited number of smaller subsystems. We continue this process until the subsystems can be realized by means of a handful of building blocks.



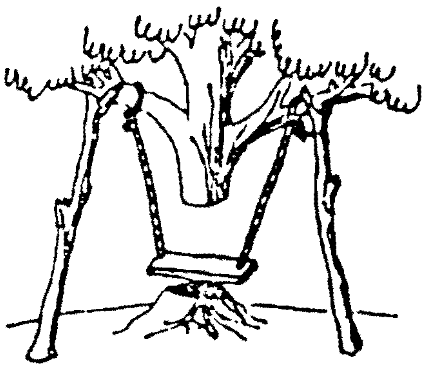
How it was designed



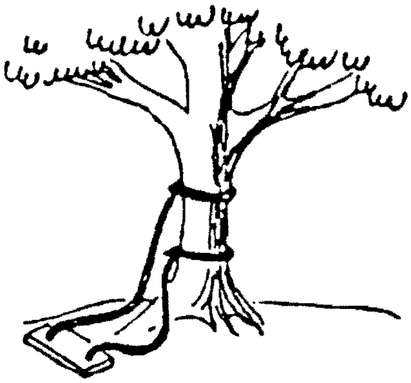
What the drawing looked like



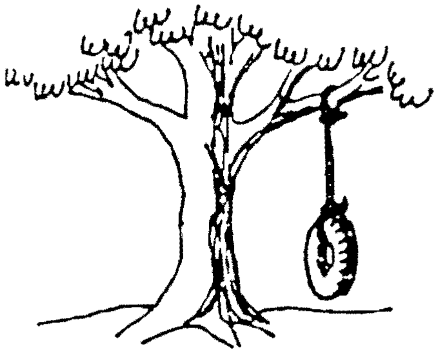
How it was ordered



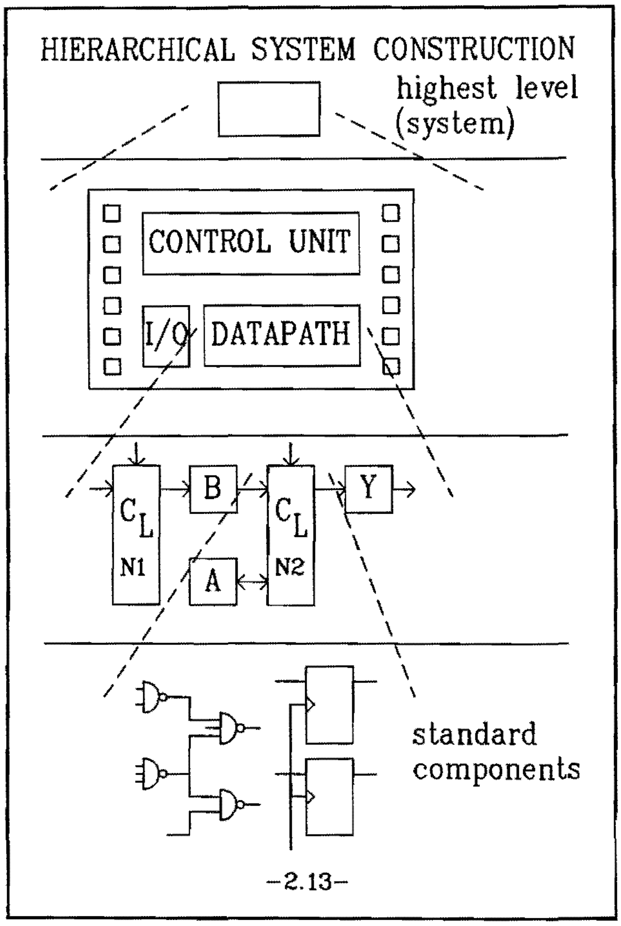
How it was mounted



How it was modified

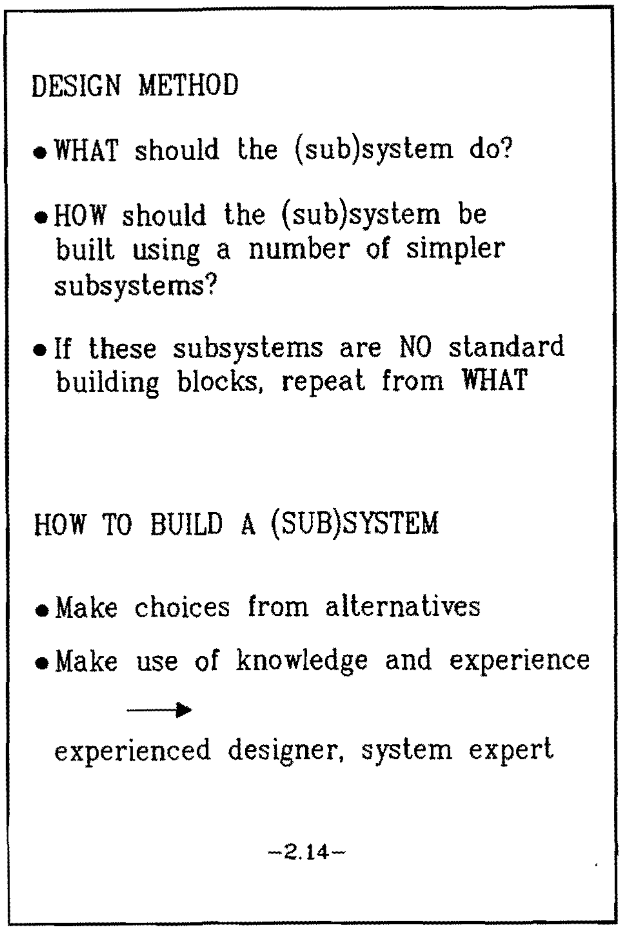


What the client really wanted



HIERARCHICAL SYSTEM CONSTRUCTION

–2.13–



DESIGN METHOD

- WHAT should the (sub)system do?
- HOW should the (sub)system be built using a number of simpler subsystems?
- If these subsystems are NO standard building blocks, repeat from WHAT

HOW TO BUILD A (SUB)SYSTEM

- Make choices from alternatives
- Make use of knowledge and experience

⟶

experienced designer, system expert

–2.14–

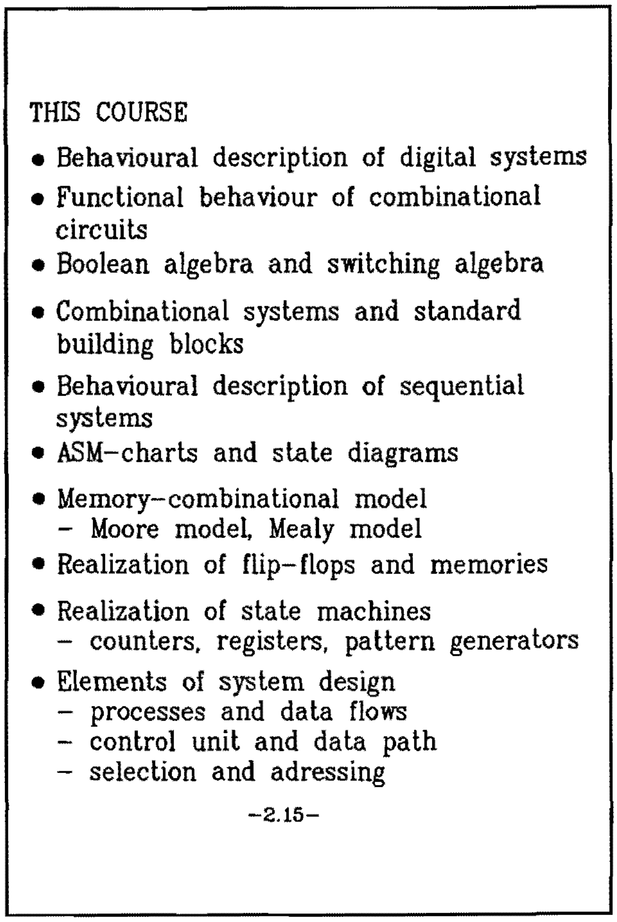
Figure 2.13 shows this hierarchical system decomposition schematically. In this case we see that the highest system level is divided into three subsystems. Each of these subsystems is internally divided into a number of smaller building blocks. Finally, we realize these building blocks with standard components. It is important to apply this structured hierarchical design methodology at all levels. That is to say, that on each level we must first describe WHAT the subsystem must do, and then, only after the system is completely specified, we should consider how to build the system from a number of limited and well-defined subsystems.

We have already divided the design process into a number of repeated applications of the following two steps (see figure 2.14):

1. What must the system do?
2. How do I build the subsystem from a number of simple blocks?

If these blocks are not standard, we should first describe what these block do, and then afterwards define how they could be realized. When building a system from a number of simple blocks we must sometimes make well balanced choices. Some alternatives may be less suitable because they lead to a less optimal realization using standard build blocks. Hence to make the choice it is necessary to have some knowledge of the set of available standard building blocks and their use in more complex modules. This is called *experience* and a person with this knowledge is an experienced designer.

Ultimately, in the future, a designer would be assisted by an expert system.



THIS COURSE

- Behavioural description of digital systems
- Functional behaviour of combinational circuits
- Boolean algebra and switching algebra
- Combinational systems and standard building blocks
- Behavioural description of sequential systems
- ASM-charts and state diagrams
- Memory-combinational model
  - Moore model, Mealy model
- Realization of flip-flops and memories
- Realization of state machines
  - counters, registers, pattern generators
- Elements of system design
  - processes and data flows
  - control unit and data path
  - selection and adressing

-2.15-

We can observe two things about the above discussion:

1. The definition of what is an optimal realization depends on many factors, such as the price, the number of components, the size of the system, the reliability, the speed, or the desired degree of security.

2. The definition of what are the standard building blocks, and consequently where the design cycle stops, depends on the state of the art. It changes towards more and more complex building blocks, and even to software automatically generating circuits (silicon compilation).

2We have now seen what digital systems are and we have discussed a method for designing them, namely the structured hierarchical design methodology. With some more knowledge about the basic building blocks, we could view this course as basically finished.

As one could expect this is not our endpoint, however. We shall consider, apart from the necessary basic knowledge and basic building blocks, the use of a design methodology.

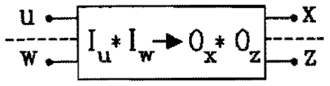
It the remainder of this course we shall first discuss (see figure 2.15) the description of digital system behaviour. Subsequently we shall study the functional behaviour of combinational systems, Boolean algebra, switching algebra, and standard building blocks related to combinational functions. Furthermore, we shall discuss the behavioral description of sequential systems, being systems with memory. Here so-called ASM charts and state diagrams play a role. A sequential system could be realized using memory and combinational functions. We shall discuss the realization of this memory using flip-flops. We shall also discuss state machines, counters, registers, pattern generators etc. And last but not least we are going to consider elements of system design such as processing and data flow, control and data path, selection and addressing.

DIGITAL SYSTEMS

- Combinational systems
- Sequential systems
- Binary systems

-2.16-

SYSTEM



- inputs

  u,..w with value set I_u,..I_w
- outputs

  x,..z with value set 0_x,..0_z
- behaviour

  - for every $u \epsilon I_u$,..$w \epsilon I_w$ there is defined a $x \epsilon 0_x$,..$z \epsilon 0_z$

$$\forall u \epsilon I_u, .. \forall w \epsilon I_w \ \exists \ x(u,..w) \epsilon 0_x$$

$$\forall u \epsilon I_u, .. \forall w \epsilon I_w \ \exists \ z(u,..w) \epsilon 0_z$$

  - if input/output relation = function:

Function F : I_u*,.. $I_w \rightarrow 0_x$*,..0_z

(X,..Z)=F(U,..W) with $u \epsilon I_u$,..$w \epsilon I_w$
$x \epsilon 0_x$,..$z \epsilon 0_z$

u,..w : independent variables

x,..z : dependent variables

-2.17-

2.3 Combinational, sequential and binary systems

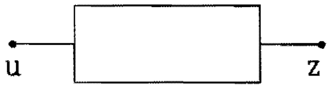
In the previous subsection we discussed the WHAT and HOW of digital system design, and we classified its internals as combinational, sequential and binary. Figure 2.17 shows what we mean by a system in this course. A system has inputs, u up to and including w, belonging to the value sets I_u up to and including I_w. Furthermore, it has outputs, x up to and including z, belonging to the value sets O_x up to and including O_z. The behaviour of such a system can be specified (defined) by a relation R. This relation R describes the relation between the input values and the output response. More formally, for all allowed input values there is at least one output value that is defined by the relation.

In this relation we consider all outputs at the same time. We can also look at every output separately. Hence, for each output, we get a relation that describes the connection between the input values and the values of the considered output. The system is described by a number of relations equal to the number of outputs. Both approaches are equally valid for our purposes. When the relation is a function (later we shall discuss what this means) we can write it in another style. We emphasize that the values of the inputs, u up to and including w, and the values of the output, x up to and including z, are defined by the system. There is a functional relation between the input and output values. We call the input and outputs of the system the *variables* of the system. u up to and including w are called the independent variables, and x up to and including z the dependent system variables.

COMBINATIONAL SYSTEM

- No memory operation
- $F : I \rightarrow 0$ is a function

This means that the response of the system to the input values are only depending on the input values at that particular moment and not on earlier input values



$u \in I \; ; \; z \in 0;$

$z = F\ (u)$

-2.18-

EXAMPLE - WEEKDAYS

$I = \{$ Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday $\}$

$0 = \{$ Yes, No $\}$

$F : z =$ <x is a weekday>
with $z \in 0 \; , \; x \in I$

EXAMPLE - COMPUTATION

$I = \{ 0 \mathinner{..} 15 \} \; ; \; 0 = \{ 0 \mathinner{..} 3 \}$

$F : z = \lfloor x/4 \rfloor$ with $z \in 0 \; , \; x \in I$

-2.19-

As previously mentioned, there are digital systems with and without memory. We shall call a system without memory a *combinational* system. Figure 2.18 shows the attributes of a combinational system. A characteristic of such a system is that the input/output relation is time invariant. The response of the system to input values depends only to the current inputs and not to the previous inputs. That is to say, the system has no memory, i.e. it cannot remember. The functional description maps the input domain uniquely to the output value domain, i.e. each input value has one specific output value. Furthermore, the function is time invariant, i.e. the mapping does not change in time, and the system always exhibits the same behaviour.

Figure 2.19 shows two examples of combinational systems. In the first example, the input set is the set of days of the week and the output set is the set of "yes"/"no". the input/output relation is given by: "is x a weekday?". It is clear that the answer to this question depends on the current input value and does not depend on previous ones. Besides the answer tomorrow will be the same as today. In the second example the variable x may take the integer values from 0 to 15. The value of the output z is given by the "floor" of x/4. The floor function has a value equal to the greatest integer being smaller or equal to its argument. Thus, the floor of 3.14 is 3 and the floor of 6.73 is 6. The value domain of z is the subset of the integer numbers from 0 to 3. This is also a system without memory.

SEQUENTIAL SYSTEM

- Memory operation
  The system remembers the input values of the past
- There is no function $F : I \rightarrow 0$
  Because the response is not only depending on the current input values but also on earlier input values

$$z = F(u_0, u_{-1}, u_{-2}, u_{-3}, \dots u_{-\infty})$$

-2.20-

EXAMPLE - COMPUTATION OF AVERAGE

$I = \{ 0 .. 15 \}$

$0 = \{ 0 .. 15 \}$

$F : z_0 = \lfloor (x_0 + x_{-1} + x_{-2} + \dots + x_{-9})/10 \rfloor$

- $z_0 \epsilon 0$, $x_i \epsilon I$
- x_{-1} .... x_{-9} are the nine previous input values

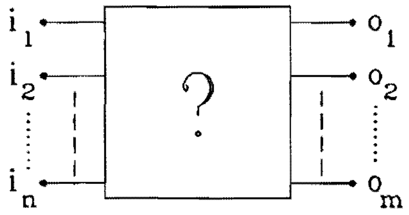
-2.21-

We shall call a system that has memory a *sequential system*. Figure 2.20 summarizes the characteristics of sequential systems. A sequential systems has a memory function, i.e. the system keeps track of previous input values. As a result, the input/output relation is not defined by a function. That is to say, the response not only depends on the current input values, but also on previous ones. A certain input value could lead to several different output values. Repeated dialling of a certain digit in a telephone number leads to changing responses from the telephone exchange. The telephone exchange remembers the previously dialled digits. In the case of sequential systems, there is a functional relation between the output z, and the current input value and all previous input values.

Figure 2.21 shows an example of a sequential system. This sequential system computes the running average of ten input values. The inputs may take integer values from 0 up to and including 15; the same holds for the outputs. The value of the output is equal to the average of the current input value and the previous nine input values. The average is rounded downwards using the floor function.

BINARY SYSTEM

- Is a digital system
- Each input and output can have two values {0,1} or {False,True} or {No,Yes}



$I = \{0,1\}^n$
- is the input set or domain
- is a set of binary n-tuples

$0 = \{0,1\}^m$
- is the output set or range
- is a set of binary m-tuples

-2.22-

A BINARY SYSTEM

- is simpler to realize than a general digital system

A DIGITAL SYSTEM

- any digital system can be replaced by an equivalent binary system

-2.23-

The binary system is a special and technically important digital system. Figure 2.22 shows some of its characteristics. A binary system is a digital system with the special characteristic that all input and output values may take only two different values. Thus every input and output is bivalent. This value set is generally given the symbols 0 and 1, false and true, real and unreal etc. In a binary system we limit the value domain of inputs and outputs. If the system has n inputs and m outputs we can say that (see figure 2.22) I is the input set of the function domain. I is formed by the set of all binary n-tuples. A binary n-tuple is a row of elements, where each element may take the values 0 or 1. Furthermore, we can define O as the output domain of the function. O is formed by the set of all binary m-tuples.

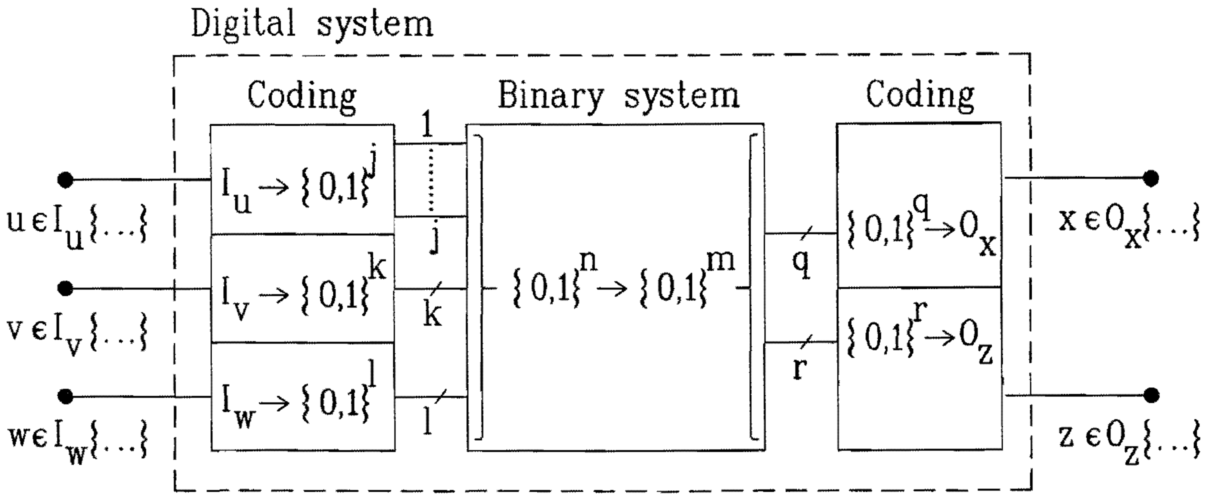
A number of advantages of binary systems are:

- The input values are bivalent. For example, this can be realized by a high and low voltage, the absence or existence of current, a high or low resistance etc. We need only to distinguish between two levels.
- Because of that a binary system is easy to realize with electronic devices (building blocks). In the realization of such a system a large inaccuracy margin may be built in.
- A realization of a binary system has a high reliability.
- A realization of binary system has generally a good resistance against possible disturbances and other external influences.

Notice that all these advantages have an influence on the realized system. Binary systems are simpler to realize than general digital systems (see figure 2.23). The specification, however - the description of the desired behaviour of a digital system - is simpler for the more general digital systems. The reason is that such systems relate better to human reasoning.

For that reason an important assertion is (see figure 2.23): *every digital system can be replaced by an equivalent binary system.*

CODING


Digital system
Coding
Binary system
Coding
$u \epsilon I_u\{...\}$
$v \epsilon I_v\{...\}$
$w \epsilon I_w\{...\}$
$I_u \rightarrow \{0,1\}^j$
$I_v \rightarrow \{0,1\}^k$
$I_w \rightarrow \{0,1\}^l$
1
j
k
l
$\{0,1\}^n \rightarrow \{0,1\}^m$
q
r
$\{0,1\}^q \rightarrow 0_x$
$\{0,1\}^r \rightarrow 0_z$
$x \epsilon 0_x\{...\}$
$z \epsilon 0_z\{...\}$

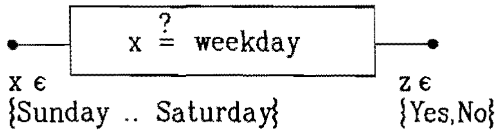
Coding = relate to each element of a set of elements a unique vector of ones and zeroes (binary vector)

-2.24-

The process of translating a digital system into a binary one is called *coding*. Figure 2.24 shows a schematic of the coding process. Coding means that every element in a set of elements is mapped to a unique vector of zeroes and ones (a binary vector). In figure 2.24 we see that every input value is translated into a binary tuple by a coding step. All these tuples together provide the binary n-tuple input to the binary system; based on this input n-tuple the system determines the m-tuple output value.

This binary value is mapped by a second coding step to elements of the digital system's output set. This second coding step is also-called the *decoding* step. Actually coding is something that can be done by a digital system, more specifically a combinational system. Usually such a coding system is not explicitly realized.

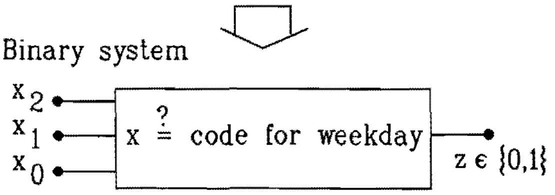
EXAMPLE - WEEKDAYS



Coding

Sunday	= 0 0 0	Thursday	= 1 0 0		
Monday	= 0 0 1	Friday	= 1 0 1	Yes	= 1
Tuesday	= 0 1 0	Saturday	= 1 1 0	No	= 0
Wednesday	= 0 1 1	unused	= 1 1 1		

Binary system



$x_2x_1x_0 \in \{0,1\}^3$

$$F = \{000 \rightarrow 0,\ 001 \rightarrow 1,\ 010 \rightarrow 1,\ 011 \rightarrow 1,\ 100 \rightarrow 1,\ 101 \rightarrow 1,\ 110 \rightarrow 0,\ 111 \rightarrow 0 \text{ or } 1\}$$

-2.25-

CODING

Coding =
give every element of a set

- a unique binary vector with a fixed length
- a fixed number of bits
- an n-tuple

There are

$$2^n$$

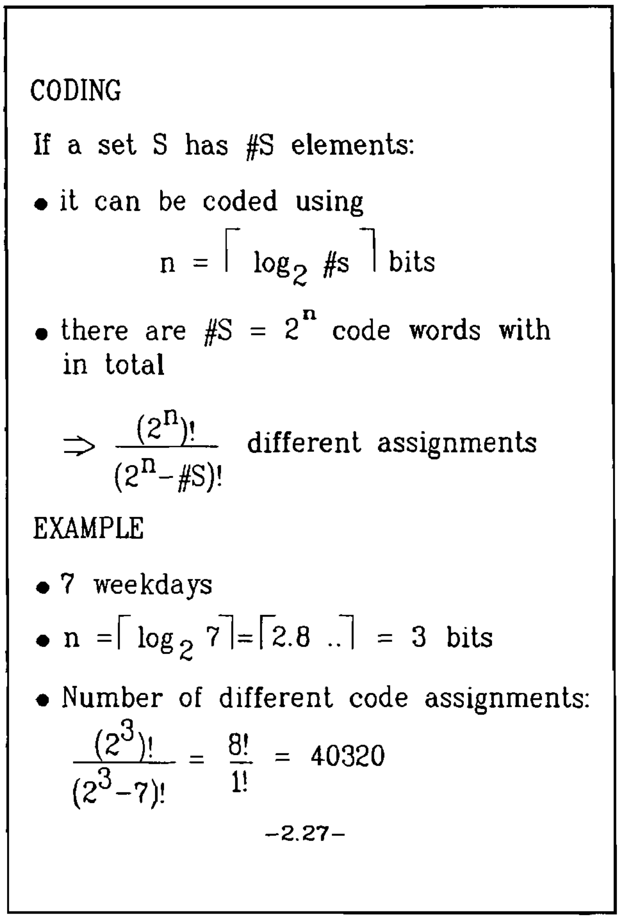
different n-bit codes (n-tuples)

- 2 bits → 4 different codes
- 3 bits → 8 different codes
- ....
- 16 bits → 65536 different codes = 64K

-2.26-

Figure 2.25 shows the first step in the process of realizing our combinational function: "Is x a weekday?". A coding step is necessary for the realization of a binary system. Figure 2.25 shows a possible coding scheme. The days of the week are coded as triples, i.e. binary vectors with three elements. Since we have three element that may take the values 0 or 1, we have eight different combinations, of which there is an unused code. In this case we use the 111 code for the element "others". Coding of the output values is simple: "yes" = 1, and "no" = 0. The equivalent binary system is given by a functional relation between the binary value on the output z and the binary values on the inputs x_0 up to and including x_2. The function can be specified to give an output value for each corresponding combination of input values. This is shown in figure 2.25. Note that for the input combination 111 the output cannot be specified; indeed 111 does not correspond to a code for a day of the week, and thus there is no answer to the question: "Is this a code for a weekday?". Later we shall return to the issue of undefined output values.

Thus coding is: giving each element of a set a unique binary vector with a fixed length, i.e. a fixed number of elements. Each element of a binary vector is also-called a binary digit or "bit". Hence an n-tuple is called an n-bit vector. A bit (a binary digit) can have two values, 0 or 1. Thus, with n bits (binary n-tuple) we can make 2^n different codes. There are four different code for two bits, and 65,536 codes for 16 bits. We also denote 65,536 by 64k, where k stands for 2^{10} = 1024.



CODING

If a set S has #S elements:

- it can be coded using

$$n = \lceil \log_2 \#s \rceil \text{ bits}$$

- there are #S = 2^n code words with in total

$$\Rightarrow \frac{(2^n)!}{(2^n-\#S)!} \text{ different assignments}$$

EXAMPLE

- 7 weekdays
- $n = \lceil \log_2 7 \rceil = \lceil 2.8 \,.. \rceil = 3$ bits
- Number of different code assignments:

$$\frac{(2^3)!}{(2^3-7)!} = \frac{8!}{1!} = 40320$$

-2.27-

Given a value set that we want to code this question arises: how many bits do we need for this coding and in how many different ways can codes be assigned. Figure 2.27 answers this question. We want to code the value set S. The number of elements in the set is given by #S. In order to code with n bits, 2^n must be at least equal to the number of elements. For that reason n must at least be equal to the "ceiling" of $\log_2(\#S)$. The ceiling function is the smallest integer grater than or equal to its arguments. With n bits we can make 2^n different code words. There are #S code words necessary for the coding. Thus 2^n-#S code words remain unused. Figure 2.27 shows how many different ways we can deduce #S code words from 2^n code words. These are the different possible code assignments.

Problem 2.1 (2.6)

Consider coding the set of integer numbers {0..99}

a. *How many bits are needed for coding this set?*
b. *How many code words are unused for this coding scheme?*
c. *How many different coding schemes can be employed?*
d. *How many elements can a set have at a maximum to enable coding with the indicated number of bits?*
e. *How many coding schemes can be employed for (d)?*

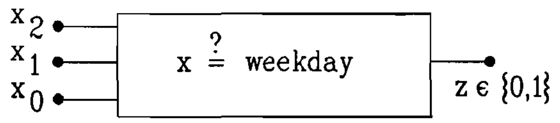
From the numeric example in figure 2.27 we see that the number of possible codes increases rapidly with n. Our seven days of the week can be coded in more than 40,000 different ways. Now the question is: are these codes equally optimal, equally good? The answer to this question is certainly *no*!. The definition of what is a good code depends on a number of factors:

- The used code must lead to a simple realization. The code must be optimal with relation to costs, reliability and availability for realization purposes.

- The used code is depending on the operation (transformation) carried out on the binary data and is depending on the way these operations are realized.

- Sometimes the use of the code is defined by an agreement, convention, or some used standard. So any two related subsystems must use the same code.

EXAMPLE - CODING

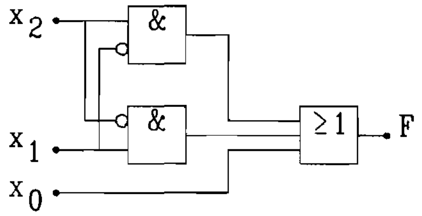


CODING 1 $(x_2 x_1 x_0)$

Sunday	= 0 0 0	Thursday	= 1 0 0	
Monday	= 0 0 1	Friday	= 1 0 1	Yes = 1
Tuesday	= 0 1 0	Saturday	= 1 1 0	No = 0
Wednesday	= 0 1 1	unused	= 1 1 1	

$$\Rightarrow \quad F = \begin{cases} 1 \text{ if } (x_0=1 \text{ or } (x_2=1 \text{ and } x_1=0) \\ \qquad\qquad \text{ or } (x_2=0 \text{ and } x_1=1) \\ 0 \text{ otherwise} \end{cases}$$

Implementation using standard building blocks



-2.28-

Figure 2.28 shows another possible coding scheme for our weekday function. We have previously specified the related binary function (see figure 2.25). We see that the output F = 1 if $x_0 = 1$ (Monday, Wednesday, and Friday) or $x_2 = 1$ and $x_1 = 0$ (Thursday, and Friday) or $x_2 = 0$ and $x_1 = 1$ (Thursday and Wednesday). Otherwise the function value equals 0. Later we shall see that this function can be realized with three standard building blocks, as shown in figure 2.28. Here we use two building blocks to realize the "and" functions. These are indicated with the & sign. A second building block is used for the realization of the "or" function. This is indicated by the ≥1 sign.

Problem 2.2 (2.7)

Consider the set {Spades, Hearts, Diamonds, Clubs}.

a. *How many bits are needed for coding this set?*

b. *How many different coding schemes are possible?*

c. - *Specify all possible coding schemes.*
   - *Are all of them really different?*
   - *Is there an essential difference between codes 01 and 10? (Remark: no weights are assigned to the individual bits).*
   - *If there is no difference: specify the number of different coding schemes.*

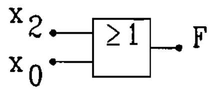
EXAMPLE - CODING

CODING 2

Sunday	= 0 0 0	Thursday	= 1 0 1	
Monday	= 0 0 1	Friday	= 1 1 1	Yes = 1
Tuesday	= 0 1 1	Saturday	= 0 1 0	No = 0
Wednesday	= 1 0 0	unused	= 1 1 0	

$$\Rightarrow \quad F = \begin{cases} 1 \text{ if } (x_2=1 \text{ or } x_0=1) \\ 0 \text{ otherwise} \end{cases}$$

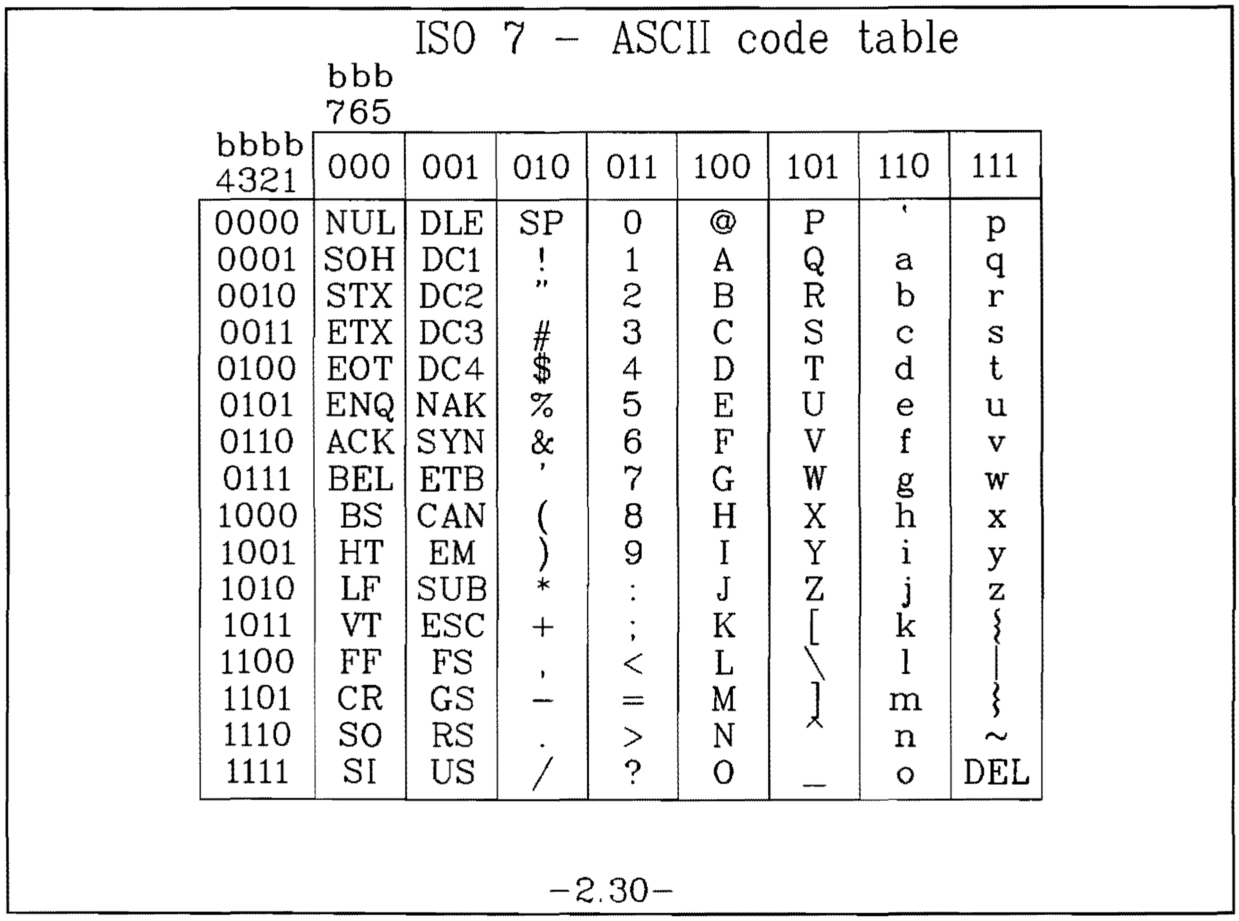
Implementation using standard building blocks



-2.29-

In figure 2.29 we see what happens if we chose another code for our function. The codes for the days of the week are chosen on purpose so that the function could be realized simply, i.e. with a minimal number of building blocks. We see that we have to deal with the code of a weekday if $x_2 = 1$ or $x_0 = 1$. This function could be realized with only one standard building block: the "or" function. We conclude that the choice of a code may have a clear influence on the complexity of the realization.

How do we chose a good code? In general this is a very difficult question; there are many possible different codes. Next it is always not clear what a good code is. In this course we shall not pay any further attention to the choice of optimal codes. However, we shall notice that a (defacto) standard code is frequently prescribed. Often the standardized ISO or ASCII codes (ASCII = American Standard Code for Information Interchange) are applied when coding the set of alphanumeric characters (letters, digits, dialectical signs etc).

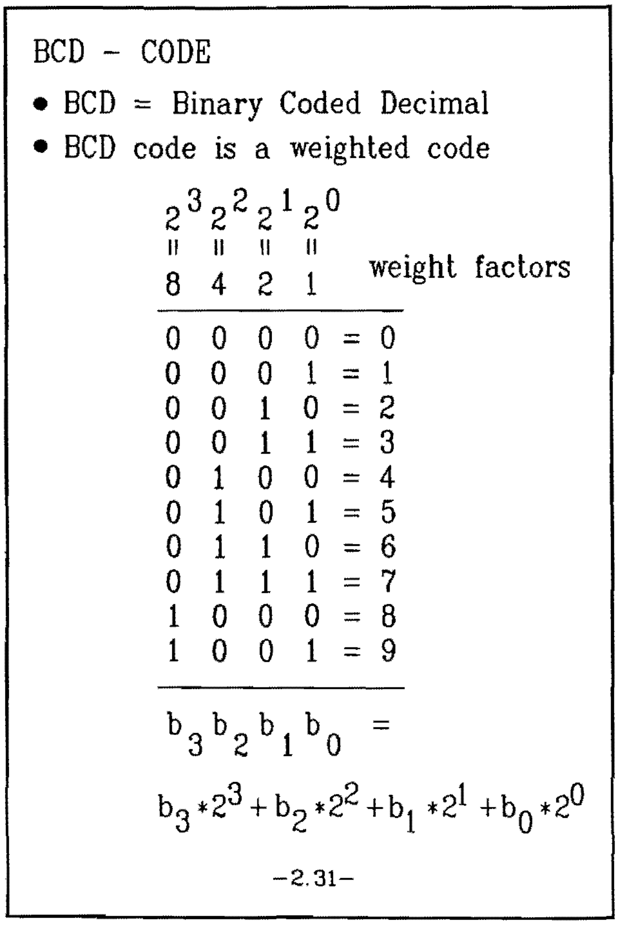


ISO 7 – ASCII code table

	bbb 765							
bbbb 4321	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	–	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

–2.30–

Figure 2.30 gives an overview of this code. We see that it contains 128 different characters. The first 31 characters are the so-called control characters. They do not belong to the set of printable characters, but are used to control printers and similar devices. There we also find codes for moving to the following line, and to place the print-head at the beginning of the line. Furthermore, we find a large number of printable characters, e.g. the digits 0 to 9 and all 26 letters of the alphabet in upper and lower case.



BCD - CODE

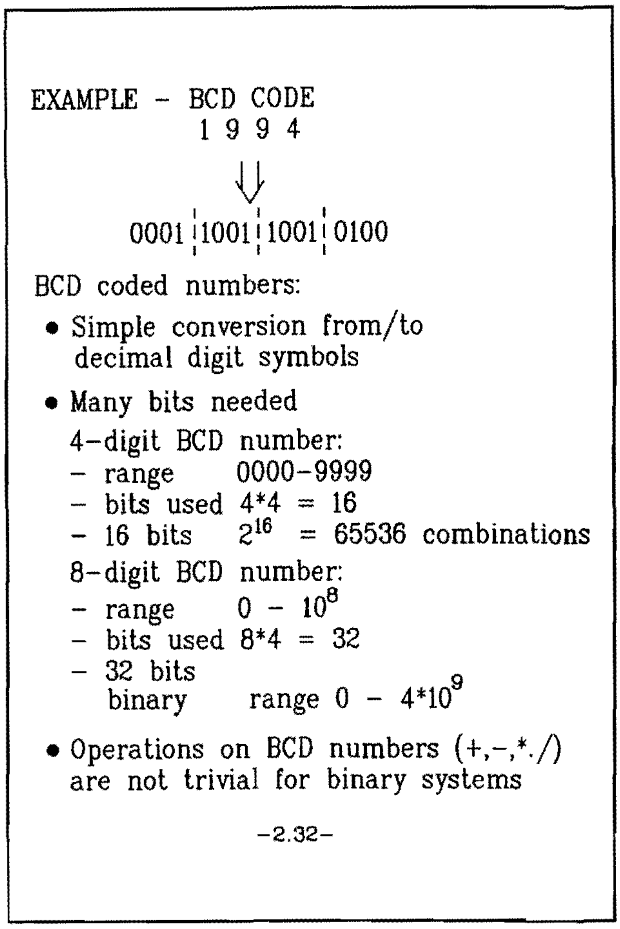
- BCD = Binary Coded Decimal
- BCD code is a weighted code

2^3	2^2	2^1	2^0	
=	=	=	=	
8	4	2	1	weight factors
0	0	0	0	= 0
0	0	0	1	= 1
0	0	1	0	= 2
0	0	1	1	= 3
0	1	0	0	= 4
0	1	0	1	= 5
0	1	1	0	= 6
0	1	1	1	= 7
1	0	0	0	= 8
1	0	0	1	= 9

$$b_3 b_2 b_1 b_0 =$$

$$b_3*2^3+b_2*2^2+b_1*2^1+b_0*2^0$$

-2.31-



EXAMPLE - BCD CODE

1 9 9 4

⇓

0001 ¦ 1001 ¦ 1001 ¦ 0100

BCD coded numbers:

- Simple conversion from/to decimal digit symbols
- Many bits needed

  4-digit BCD number:
  - range 0000-9999
  - bits used 4*4 = 16
  - 16 bits 2^{16} = 65536 combinations

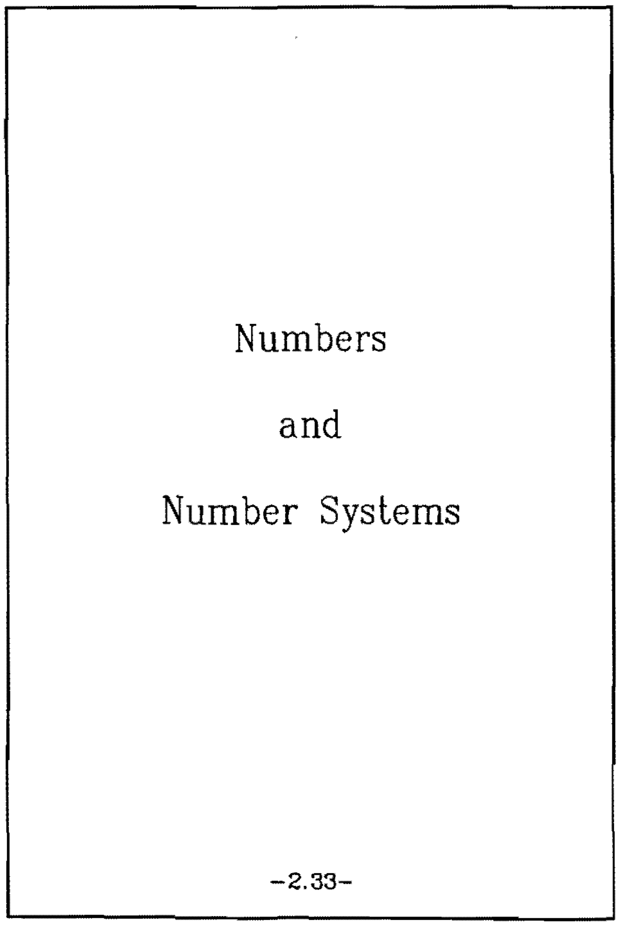
  8-digit BCD number:
  - range 0 - 10^8
  - bits used 8*4 = 32
  - 32 bits
    binary range 0 - $4*10^9$
- Operations on BCD numbers (+,-,*,/) are not trivial for binary systems

-2.32-

Another frequently used standard code is the BCD code. BCD stands for Binary Coded Decimal. These codes can be used to code decimal digits into numbers. Figure 2.31 gives an overview. The BCD code is a weighted code. That is, each individual bit position is associated with a weight factor. In the BCD code these weights are, from right to left, 2^0 = 1, 2^1 = 2, 2^2 = 4, and 2^3 = 8. Using these weight factors we can calculate the decimal equivalent of a BCD coded digit. This is done by multiplying each bit value (0 or 1) by the corresponding weight factor, and then summing up all the results. Numbers can be coded into BCD by translating each individual digit into its equivalent BCD code.

Figure 2.32 shows how to code the number 1994. We see how individual digits are translated into a 4-bit BCD code.

The conversion between BCD coded numbers and decimal numbers is simple. In reality BCD code has some disadvantages. First we name the small domain. We need 4x4 = 16 bits to code a number with four decimal digits, i.e. 0..9999. But as mentioned earlier 16 bits provide 65,536 possibilities. Thus the numbers 0 to about 65,000 can be coded in 16 bits. When considering eight digits the difference between BCD coding and the number of different codes becomes much larger. A second disadvantage of BCD codes is that the standard operations on numbers such as addition, subtraction, multiplication and division of BCD coded numbers are not easy to realize in a binary system. Consequently binary numbers often constitute a better choice.

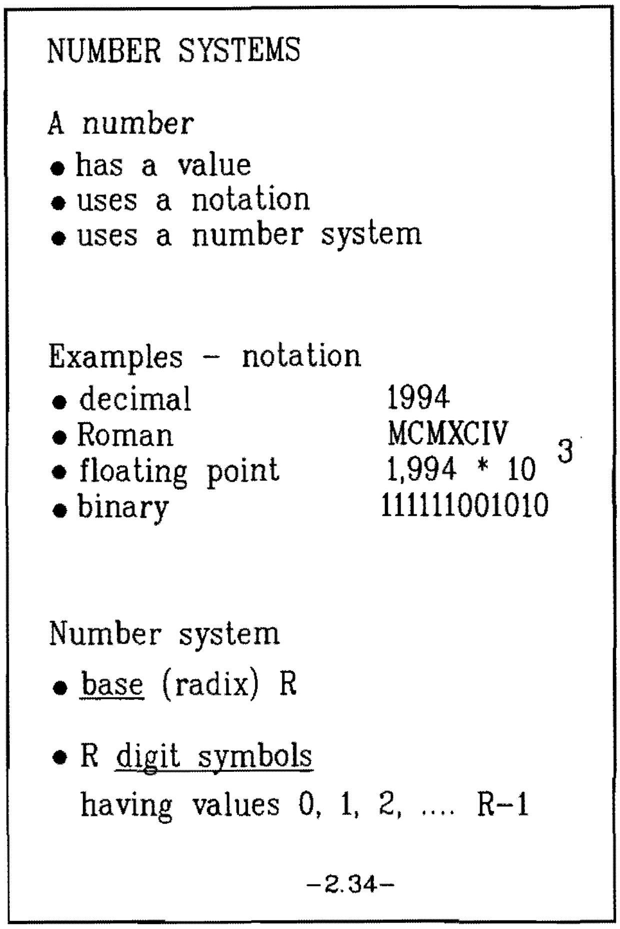


Numbers

and

Number Systems

-2.33-



NUMBER SYSTEMS

A number
- has a value
- uses a notation
- uses a number system

Examples – notation

● decimal	1994
● Roman	MCMXCIV
● floating point	$1,994 * 10^3$
● binary	111111001010

Number system
- base (radix) R
- R digit symbols
  having values 0, 1, 2, ... R–1

-2.34-

2.4 Numbers and number systems

Now, we shall briefly overview numbers and how they can be represented. We shall consider number systems. We must remember that a number has a value, and besides that, has a notational form or way of representation. The value of a number is fixed; the method of writing it mostly depends on the number system in which this value is given. Figure 2.34 demonstrates this by displaying a specific number, 1994, in four different ways. First we see the familiar decimal representation. Next, the same number is shown as the Romans would write it using the Roman digit symbols. The next representation is given in the so-called floating point notation. And finally we have the binary representation of the number. Each notation has a set of conventions, or rules that are used to calculate the value of the number. Such a set of conventions or rules is called a *number system.* A number system is formed by a base or radix with a value R, and in total R digit symbols with values 0, 1, 2, .. R-1, respectively.

Number systems

Decimal number system

- $R = 10$
- $d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- notation : 1994
- value = $1*10^3+9*10^2+9*10^1+4*10^0$

Binary number system

- $R = 2_{10}$
- $d_i \in \{0,1\}$
- notation: 10011
- value = $1*2^4+0*2^3+0*2^2+1*2^1+1*2^0$

Hexadecimal number system

- $R = 16_{10}$
- $d_i \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$
- notation: 9EA3
- value = $9*16^3+E*16^2+A*16^1+3*16^0$

-2.35-

NOTATION AND VALUE

Number notation

- n-tuple of digit symbols
  $d_{n-1}\, d_{n-2} \,.....\, d_2\, d_1\, d_0$
- $d_i \in \{\text{digit symbols}\}$

Number value

$$\text{value} = \sum_{i=0}^{n-1} d_i * R^i$$

$$= d_{n-1}R^{n-1}+d_{n-2}R^{n-2}+...+d_2R^2+d_1R^1+d_0R^0$$

-2.36-

Figure 2.35 shows some examples of number systems. The first example is the decimal system. In this system the base value (radix) is 10. The digit symbols are formed by the digits 0 to 9. Also we see how the value 1994 is denoted in the decimal system. The value can be computed by considering each digit position as coupled to a weight factor. The rightmost digit has the weight 10^0, the next digit has a weight factor 10^1, then 10^2, and finally the weight factor 10^3. The weight factor is thus a power of the base.

In the binary number system the base is equal to 2. The digits are formed by the symbols 0 and 1. A binary number is formed by a row (n-tuple) of binary digits, or bits. Figure 2.35 shows an example of a binary number. The value of a binary number can be defined by multiplying the individual binary digits by the corresponding weight factors, and then summing up the results. The weight factors are, once more, powers of the base, i.e. powers of 2. Figure 2.35 shows this value computation, expressed in the decimal system. A third important number system is the hexadecimal system or the 16-digit system. In this system the base value is 16. The digit symbols are now formed by the digits from 0 to 9 and, because we need 16 symbols, the letters from A to F. The symbols A to F have the values 10 to 15, respectively. Figure 2.35 shows an example of the notational form and the value computations expressed in the decimal system.

Figure 2.36 shows that the notational form of a number is an n-tuple of digit symbols, each of these digit symbols being selected from the set of R (Radix) digit symbols. The value of a number is defined by the weighted sum of all digit symbols, the weight factor being a power of the base. We notice that the determination of this value (the calculation of the weighted sum) can be performed in different number systems. Because we humans find it easier to calculate in the decimal system, we shall generally determine its value using computations in decimal. In a computer it is easier to use a binary system.

Hence the use of different systems gives different notational forms for the same number. We are interested in methods to convert a number from one notational form to another, preserving its value. There are two available methods. The methods we use depend on:

- the number system in which the calculation will be executed;
- the original number system in which the number is represented;
- the target number system in which we want to represent the number.

NUMBER SYSTEM CONVERSION

Base x ⟶ base 10

- Computation in decimal
- Method: repeated <u>multiplication</u>
- Number value =

$$(..(d_{n-1}*R+d_{n-2})*R+...+d_2)*R+d_1)*R+d_0$$

Example – binary to decimal

$$1010_2 = ((1*2+0)*2+1)*2+0 = 10_{10}$$

$$110110_2 = ((((1*2+1)*2+0)*2+1)*2+1)*2+0 = 54_{10}$$

Alternative method:

$$110110 = 1*2^5+1*2^4+0*2^3+1*2^2+1*2^1+0*2^0$$
$$= 32 + 16 + 4 + 2 = 54_{10}$$

Example – hexadecimal to decimal

$$9EA3 = ((9*16+14)*16+10)*16+3 = 40611_{10}$$

$$7CA = (7*16+12)*16+10 = 1994_{10}$$

Alternative method:

$$7CA = 7*16^2+12*16^1+10*16^0 =$$
$$= 1792 + 192 + 10 = 1994_{10}$$

-2.37-

Problem 2.3 (2.4)

Consider the binary number system using n bits for the representation of numbers.

a. Show that the greatest value that can be represented is:
$2^n - 1$

b. What is the greatest value for n=8?

c. What is the greatest value for n=16?

We apply repeated multiplications if the system we shall use for calculations and the target system are equal. Figure 2.37 shows an example of a conversion to the decimal system with the calculations also carried out in the decimal system. We begin by noting the value of the most significant digit symbol, i.e. the leftmost one. When there are more digits in the number, we multiply this decimal value with the base where this number is coded. This multiplication is carried out in the decimal system. Subsequently we add the result to the value of the following digit symbol. If there are more digits we repeat the whole procedure. The final result of this calculation is the value of the number in decimal. Figure 2.37 shows a number of conversion examples from the binary and hexadecimal systems.

NUMBER SYSTEM CONVERSION

Base 10 ⟶ base x

- Computation in decimal
- Method: repeated <u>division</u>
- Determination of digits:

Digit 0 (rightmost digit):

$$\left(\sum_{i=0}^{n-1} d_i * R^i\right) \div R = \sum_{i=1}^{n-1} d_i * R^{i-1} \text{ remainder } d_0$$

Digit k:

$$\left(\sum_{i=k}^{n-1} d_i * R^{i-k}\right) \div R = \sum_{i=k+1}^{n-1} d_i * R^{i-k-1} \text{ remainder } d_k$$

Example – decimal to binary

83 ÷ 2 = 41 rem 1 least sign. bit
41 ÷ 2 = 20 rem 1
20 ÷ 2 = 10 rem 0
10 ÷ 2 = 5 rem 0
5 ÷ 2 = 2 rem 1
2 ÷ 2 = 1 rem 0
1 ÷ 2 = 0 rem 1 most sign. bit

$$83_{10} = 1010011_2$$

Example – decimal to hexadecimal

$2783_{10} \div 16 = 173$ remainder 15
$173 \div 16 = 10$ remainder 13
$10 \div 16 = 0$ remainder 10

$$2783_{10} = ADF_{16}$$

-2.38-

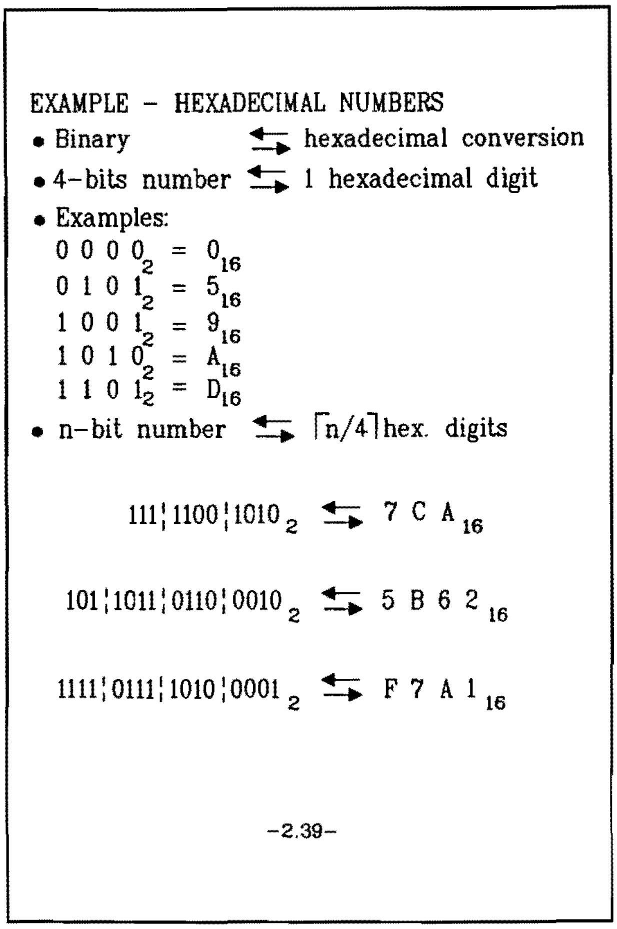
Problem 2.4 (2.3)

Show that:

a. $10110101_2 = 181_{10}$

b. $01101100_2 = 108_{10}$

The second method is applied if the number system used for the calculation is equal to the number system used to code the number. We make use of the method of repeated division. Figure 2.38 shows how to convert a number from the decimal system by means of calculations also executed in the decimal system. In this method of repeated division we divide the number by the base of the target number system. In figure 2.36 we see that if we perform such a division we keep a remainder that is equal to the value of the least significant, i.e. the rightmost, digit. If we divide the result again by the base of the target system then we get a remainder equal to the next digit. By repeating this division we translate all the digits of the number into the target system, starting with the least significant digit and ending with the most significant one. Figure 2.38 shows two examples: in the first example a number is converted into binary and in the second one into hexadecimal.



EXAMPLE - HEXADECIMAL NUMBERS

- Binary $\leftrightarrows$ hexadecimal conversion
- 4-bits number $\leftrightarrows$ 1 hexadecimal digit
- Examples:

$0\,0\,0\,0_2 = 0_{16}$
$0\,1\,0\,1_2 = 5_{16}$
$1\,0\,0\,1_2 = 9_{16}$
$1\,0\,1\,0_2 = A_{16}$
$1\,1\,0\,1_2 = D_{16}$

- n-bit number $\leftrightarrows$ $\lceil n/4 \rceil$hex. digits

$111|1100|1010_2 \leftrightarrows 7\ C\ A_{16}$

$101|1011|0110|0010_2 \leftrightarrows 5\ B\ 6\ 2_{16}$

$1111|0111|1010|0001_2 \leftrightarrows F\ 7\ A\ 1_{16}$

-2.39-

Problem 2.5 (2.2)

Show that:

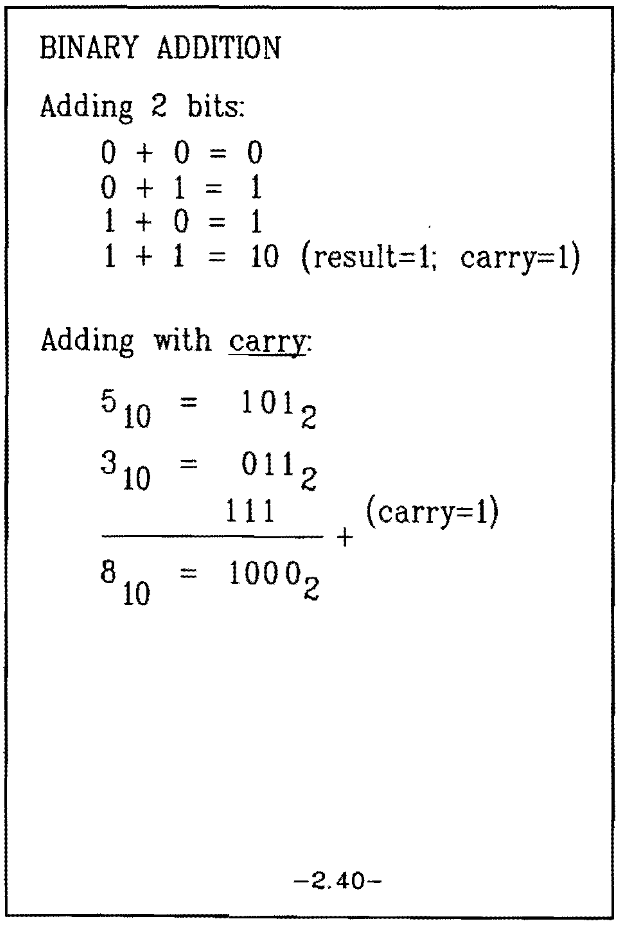
a. $216_{10}=3120_4$
b. $99_{10}=1203_4$
c. $A7_{16}=2213_4$
d. $83_{16}=2003_4$

Binary systems will generally use numbers denoted in the binary system. A disadvantage of using binary numbers is that the number of digit symbols (bits) is large, even for relatively small numbers. We need ten digit symbols to represent the decimal number 1000. The advantage of the hexadecimal system is that it provides a very compact way of writing numbers, and at the same time, it is simple to convert hexadecimal numbers to and from the binary system. Hence hexadecimal numbers are frequently used as a short way of writing binary numbers.

Problem 2.6 (2.5)

a. Write 10985_{10} *in the binary system.*
b. Write 278_{10} *in the base 5 number system.*

Figure 2.39 discusses the conversion between binary and hexadecimal numbers, being a very simple one: a four bit binary number can be coded into a single hexadecimal digit. Indeed, with four bits we can have 16 different quadruples which can clearly be represented by the 16 digits of the hexadecimal system. If you wish you can verify this representation by converting the binary and hexadecimal numbers to the decimal system. The opposite way is also possible. A hexadecimal digit can always be written as a 4-digit binary number. And an n-bit binary number can be converted to (n/4)-digit hexadecimal number, beginning with the least significant bit, forming groups of four bits, and replacing these groups by the corresponding hexadecimal digit. Figure 2.39 shows a number of examples.



BINARY ADDITION

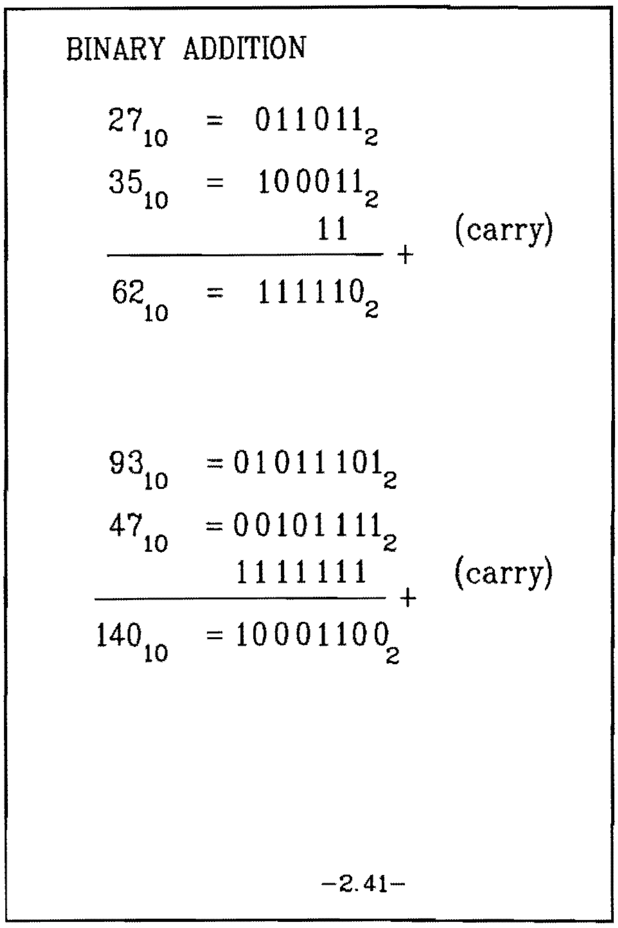
Adding 2 bits:

$$\begin{array}{l} 0 + 0 = 0 \\ 0 + 1 = 1 \\ 1 + 0 = 1 \\ 1 + 1 = 10 \text{ (result=1; carry=1)} \end{array}$$

Adding with carry:

$$\begin{array}{rcrl} 5_{10} & = & 101_2 & \\ 3_{10} & = & 011_2 & \\ & & 111 & + \text{ (carry=1)} \\ \hline 8_{10} & = & 1000_2 & \end{array}$$

-2.40-



BINARY ADDITION

$$\begin{array}{rcrl} 27_{10} & = & 011011_2 & \\ 35_{10} & = & 100011_2 & \\ & & 11 & + \quad \text{(carry)} \\ \hline 62_{10} & = & 111110_2 & \end{array}$$

$$\begin{array}{rcrl} 93_{10} & = & 01011101_2 & \\ 47_{10} & = & 00101111_2 & \\ & & 1111111 & + \quad \text{(carry)} \\ \hline 140_{10} & = & 10001100_2 & \end{array}$$

-2.41-

We have now seen how to denote numbers in the binary system. The following step is to be able to calculate with binary numbers. In this course we shall limit ourselves to addition and multiplication of two positive binary numbers. When we learn to add in the decimal system we could start by adding two digits. Applying this to the binary system means adding two binary digits, i.e. two bits. Figure 2.40 shows the rules for adding two bits.

Regarding calculations in the decimal system, when adding two digits the result is sometimes too large to be represented in one digit. For example 9 + 7 gives 16. We say then 9 + 7 = 6 with a "carry" of 1. Something similar is valid for the binary system. We see that if we calculate 1 + 1 the result is 0 with a carry of 1. This carry is used when adding the next two digits in the number as shown in figure 2.40. There the carry from the last (right) two digits is explicitly shown.

Figure 2.41 shows another two examples. We recommend that you carefully study these two examples and that you practice with the addition of other binary numbers.

Problem 2.7 (2.1)
Determine the sum of the binary numbers 011011 and 110110. What do you notice concerning the number of bits in the result?

BINARY MULTIPLICATION

$$\begin{array}{rcr l}
18_{10} & & 10010 & \text{(multiplicand)} \\
103_{10} & & 1100111 & \text{(multiplier)} \\
\hline
 & & & \text{x} \\
1 * 18 & = & 10010 & \\
1 * 18 & = & 10010\cdot & \\
1 * 18 & = & 10010\cdot\cdot & \\
0 * 18 & = & 00000\cdot\cdot\cdot & \\
0 * 18 & = & 00000\cdot\cdot\cdot\cdot & \\
1 * 18 & = & 10010\cdot\cdot\cdot\cdot\cdot & \\
1 * 18 & = & 10010\cdot\cdot\cdot\cdot\cdot\cdot & \\
\hline
 & & & + \\
1854_{10} & = & 11100111110_2 &
\end{array}$$

Multiplication by a power of 2

- $* 2_{10}$ = shift left 1 position:

  $d_{n-1}d_{n-2}\cdots d_2d_1d_0 * 10_2 = d_{n-1}d_{n-2}\cdots d_2d_1d_0 0$

  $10110_2 * 10_2 = 101100_2$

- $* 2^n{}_{10}$ = shift left n positions:

  $1000_2 * 1000_2 = 1000000_2$

-2.42-

NOT COVERED IN THIS COURSE

- Negative numbers
- Fractional numbers
- Floating point numbers
- Subtraction and division
- Finite register length
- Error detecting and error correcting codes

-2.43-

When considering multiplication in the binary system we could also look at related examples in the decimal system. Also in multiplication the multiplicand is successively multiplied by all digits in the multiplier starting with the least significant (i.e. the rightmost) digit. Now the digits of the multiplier have the value 1 or 0. The result of the multiplication is consequently the multiplicand itself or 0. As usual, the intermediate result is shifted one place to the left and then we add all these intermediate results. This gives the result of the multiplication. Figure 2.42 shows a worked out example.

Furthermore we shall pay attention to a special attribute of binary multiplication. It appears that binary multiplication of a number by 2 is equivalent to shifting the binary number one position to the left and adding a least significant 0 to the right of the number. Figure 2.42 shows an example. We also observe that a multiplication by 2^n (2, 4, 8, 16 etc.) means that the binary number is shifted to the left by n positions and the free positions are filled with zeros. These characteristics are frequently utilized in digital systems.

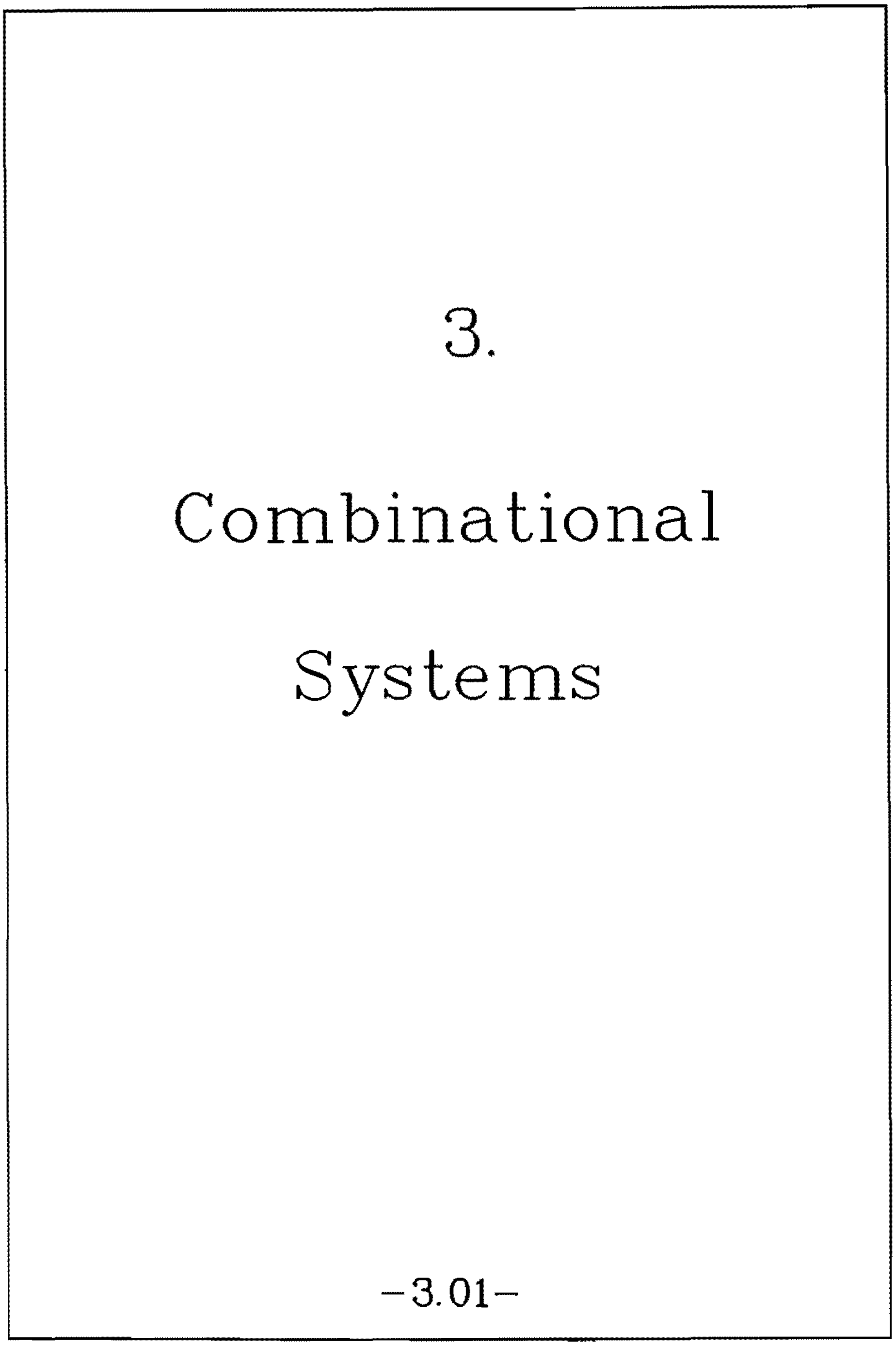
Thus far we have given a short introduction to binary numbers and binary calculations. We are aware that a lot of topics remain untouched. An overview is given in figure 2.43. We did not go into the representation of negative numbers or fractions. We did not consider floating point notations. Neither did we consider binary subtraction or division. Also we did not cover problems arising from the fact that in binary systems numbers are represented by a fixed number of bits. We only mention the existence of other codes, such as error detecting and correcting codes.

2.5 Summary

We have tried to answer the questions: "What is a digital system?", and "How do I design a digital system?". As an important criterion of digital systems we have discussed inputs and outputs being elements of finite value sets. We have discussed behavioural descriptions by means of a functional relation, referred to as the black box model, or by means of a number of transformation rules or algorithms: the algorithmic model. When designing a digital system it is important to first determine what we want to realize. We must make a clear behavioural description. Furthermore we must structure the design to manage its complexity. We must follow a structured design methodology yielding a hierarchical system. This is called a structured hierarchical design methodology.

Furthermore we have seen that digital systems could be divided into two classes: systems with and without memory functions. Systems without memory functions are called combinational systems, and systems with memory are called sequential systems. We have also observed that digital systems are generally realized as binary systems. A binary system is a digital system, its inputs and outputs having two different values.

The process used to convert general digital systems to binary systems is called coding. We have seen that we could code in several ways and that not all codes give equally good results. Next, we have seen that the choice of code is frequently imposed by standards, or as a result of working with other subsystems. The topic of number systems is closely related to codes. In this course we shall use three different number systems: the decimal, the hexadecimal and the binary systems. Binary systems are generally suited for manipulating and calculating with binary numbers. In this course we have limited ourselves to dealing with addition and multiplication of two positive binary numbers.



3.

Combinational

Systems

-3.01-

3 Combinational systems

We have seen in the previous chapter that a combinational system is a system without memory. The behaviour of the system can be described by a functional relation between the input and output values. We can say that the system maps values in the input domain to values in the output domain. This mapping scheme does not change in time.
In the coming three subsections we shall consider the design and realization of combinational systems.

We shall apply a structured hierarchical design methodology. We first describe what the system must do; afterwards we try to realize the system by combining a limited number of subsystems. In this chapter we shall first consider the behavioural description of systems, i.e. what the system must do. Subsequently we shall discuss two standard methodologies for decomposing a system into several subsystems.

COMBINATIONAL SYSTEM

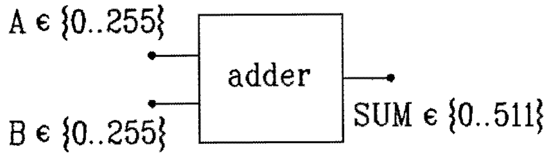
- Transformation
  system domain ⟶ system range
- Domain and range:
  finite sets

EXAMPLES

- {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}
- {0..255}

-3.02-

COMBINATIONAL SYSTEM
Example 1 – Adder



```
VAR A,B:  0..255;
     SUM:  0..511;
BEGIN
   SUM := A + B
END
```

-3.03-

3.1 Behavioural description; specification

As a combinational system becomes more complex we prefer to use the algorithmic approach (model) to describe its behaviour. We shall still use the black box model for functions that are simpler to describe. In figure 3.02 we see that the behaviour of the combinational system can be determined by a transformation of the system domain to the system range. The domain (the set of input values) and the range (the set of output values) are finite sets. A description of the system behaviour always begins with the definition of the system's domain and range. We shall denote these sets in two different ways. First we can name all elements of the set. Figure 3.02 shows such a definition for the days of the week example. We also shall specify a set by placing the first and last elements, separated by two dots, between braces. Figure 3.02 shows such a definition of the set of integer numbers from 0 to 255.

When writing an algorithmic behavioural description we shall make use of the syntax and semantics of the Pascal language.

We shall make use of the operations defined in this language, and by doing so we shall learn what these operations do and mean.

As mentioned earlier we shall use a similar language to define the behaviour of systems, i.e. to specify them. To explain how this works we shall consider a few examples. Figure 3.03 describes the behaviour of an adder. The adder is system with two inputs and one output. Both inputs can take integer values from 0 to 255. The system adds both values and the result appears on the output. The output therefore has a value range from 0 to 511. However, this last value can never occur as a result of an addition. The reason for choosing 511 as the upper limit will be explained later. The system's behaviour is explained in one Pascal statement: SUM becomes A + B. Notice that from the system we have determined the input domain, the output range and the transformation from the input domain to the output range. Therefore, the behaviour is sufficiently specified.

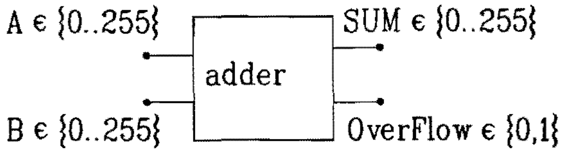
Problem 3.1
Consider a system with:

inputs	*A,B*	*∈ {0..15}*
output	*MUL*	*∈ {0..255}*
function	*F:*	*MUL = A x B*

Make a behavioural description in Pascal.

COMBINATIONAL SYSTEM

Example 2 -
Adder with A,B,SUM ∈ {0..255}



```
VAR A,B,SUM: 0..255;
    OverFlow: 0..1;

BEGIN
   SUM := (A+B) MOD 256;
   OverFlow := (A+B) DIV 256
END
```

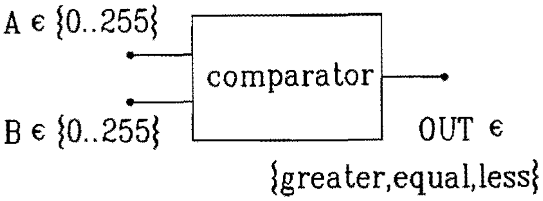
-3.04-

We often need adders where the output set is the same as both input sets. This enables us to use the sum output as an operand in a new addition. Figure 3.04 shows a description of such an adder. We see that the value range of the variables A, B and SUM are integer numbers from 0 to 255. Because the addition of such numbers could produce a result lying outside the value range of the sum, we need a second output from our system. The second output indicates whether the result is in the range of SUM. This extra output is called overflow. Naturally, this output may take two values. It will take the value no if the result of SUM is not outside the range, and yes if the result of SUM is in the range. Both possibilities are given by "overflow = 0" or "overflow = 1."

It should be clear by now, from the behavioural description, that the range of the output SUM is limited. Therefore, we can make use of the modulo operator. The behavioural description, shown in figure 3.04, uses the construct A + B MOD 256. This construct will always yield an output value between 0 and 255, i.e. if necessary the number 256 will be subtracted from A + B. The value of the overflow is determined by dividing A + B by 256 and then rounding the result downwardly (i.e. truncating). This is done by the DIV operator.

COMBINATIONAL SYSTEM

Example 3 - Comparator



```
VAR A,B: 0..255;
    OUT: (greater,equal,less);
BEGIN
   IF (A > B)
   THEN OUT := greater
   ELSE IF (A=B)
        THEN OUT := equal
        ELSE OUT := less
END
```

-3.05-

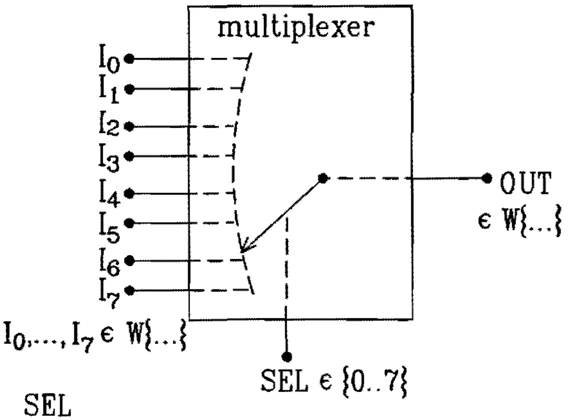
If the sum of A and B lies in the range from 0 to 255, then the overflow will be equal to 0, otherwise it will be equal to 1. Operators, such as MOD and DIV, describe a behaviour and do not suggest any details about their realization.

As a third example, figure 3.05 describes a system that compares two numbers lying within the value range from 0 to 255. The output shows the result of this comparison. The output may take the values "greater", "equal" and "smaller". We shall call such a system a comparator. When describing the behaviour of this system the use of operators alone is not sufficient; we shall need to use another construct. We have to compare the values of both inputs and based on that take an action. Such a behaviour can be described with:

if A > B then OUT equals greater.

We can describe the other cases similarly.

COMBINATIONAL SYSTEM
Example 4 – Multiplexer



SEL

- is a control (selection) input
- determines the <u>function</u> of OUT
- does <u>not</u> determine the <u>value</u> of OUT

$$F : SEL \rightarrow \{I_{d,SEL} \mid 0 \leq SEL \leq 7\}$$

$$\text{with } OUT = I_d(I_{SEL}) = I_{SEL}$$

$$OUT,\ I_{SEL} \in W\{\ldots\}, SEL \in \{0..7\}$$

-3.06-

COMBINATIONAL SYSTEM – PASCAL DESCRIPTION
Example 4 – Multiplexer; IF statement

```
VAR I0,I1,I2,I3,I4,I5,I6,I7,OUT: AnyType;
    SEL: 0..7;
BEGIN
  IF (SEL = 0) THEN OUT :=I0;
  IF (SEL = 1) THEN OUT :=I1;
  IF (SEL = 2) THEN OUT :=I2;
  IF (SEL = 3) THEN OUT :=I3;
  IF (SEL = 4) THEN OUT :=I4;
  IF (SEL = 5) THEN OUT :=I5;
  IF (SEL = 6) THEN OUT :=I6;
  IF (SEL = 7) THEN OUT :=I7
END
```

-3.07-

As a fourth example, figure 3.06 discusses the behaviour of a multiplexer. A multiplexer is the digital system equivalent to a multiple position switch. Figure 3.06 shows an 8-input multiplexer. Besides these 8 inputs I_0 to I_7, the multiplexer contains the so-called *selection input* SEL. The value of this selection input (in our case between 0 and 7) determines the input to be connected to the output. We also can say that the value of the selection input determines the function of the multiplexer, and that it only indirectly determines the value of the output OUT. These thoughts lead us to the functional description shown in figure 3.06. There the multiplexer is defined by reflecting the value of the input SEL to a set of identity functions I_d. Each of these identity functions, $I_{d,SEL}$, maps the selected input, I_{SEL}, to the output. Note that we limit ourselves to the situation where all inputs I_{SEL} and the output OUT have the same value set. This restriction does not have to be made formal, but in practical implementations of multiplexers this is always the case.

Notice also that the exact nature of the input and output value sets is not really important. This fact is also shown in the Pascal description of the behaviour shown in figure 3.07. There we have defined the variables I_0, I_1, .. I_7, OUT of type "AnyType". Furthermore, we see that the behavioural description of the multiplexer only contains some "IF THEN" statements; a statement for each possible value of the selection. Notice that one of the eight possible conditions will be true and that only one related "THEN clauses" will be executed. Thus, the output gets its correct value.

Problem 3.2

Consider a system with:

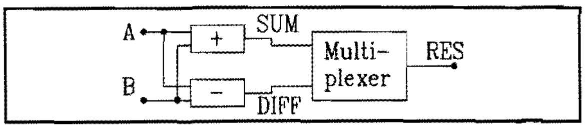
inputs	*A,B*	*$\subseteq$ {-9..9}*
	SEL	*$\subseteq$ {Add, Subtract}*
output		*RES $\subseteq$ {-19..19}*
function	*F:*	*RES = A+B if SEL = Add*
		RES = A-B if SEL = Subtract

Make a behavioural description in Pascal.

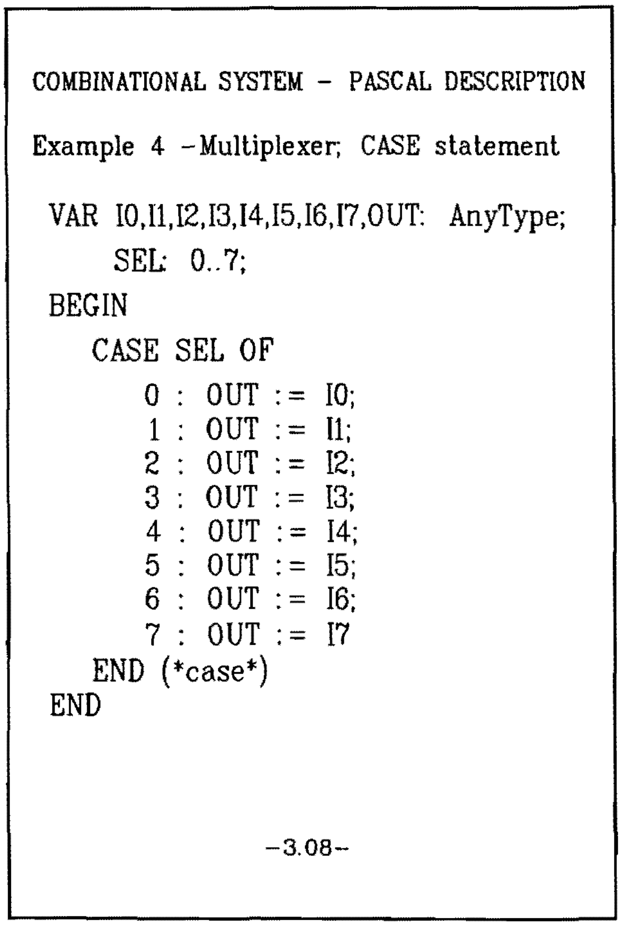
Problem 3.3

Consider the system given in problem 3.2.

The following architecture is given for the realization:



Make a behavioural description in Pascal for the system parts and the whole system.

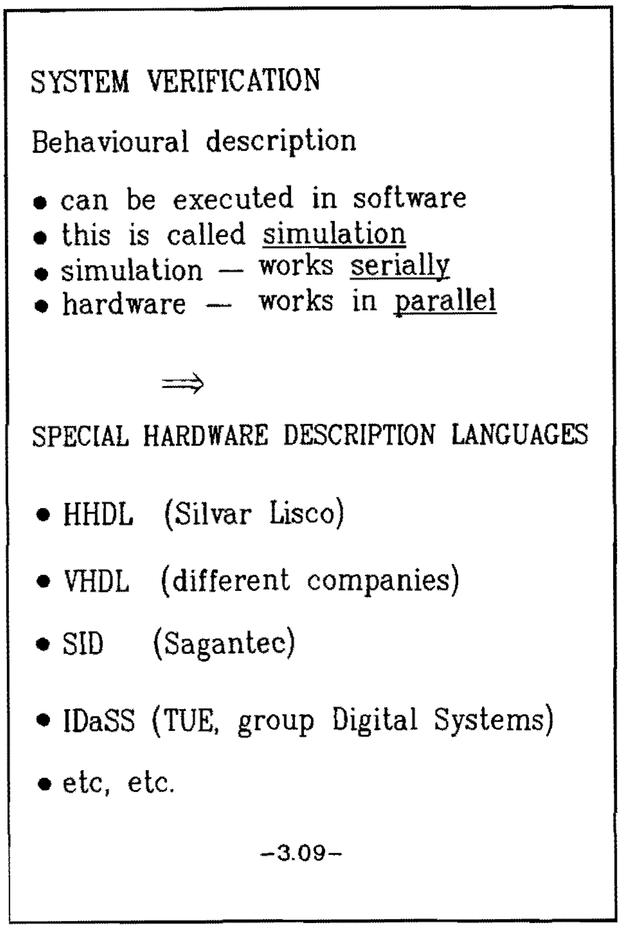


COMBINATIONAL SYSTEM - PASCAL DESCRIPTION

Example 4 -Multiplexer; CASE statement

```
 VAR I0,I1,I2,I3,I4,I5,I6,I7,OUT: AnyType;
     SEL: 0..7;
 BEGIN
    CASE SEL OF
       0 : OUT := I0;
       1 : OUT := I1;
       2 : OUT := I2;
       3 : OUT := I3;
       4 : OUT := I4;
       5 : OUT := I5;
       6 : OUT := I6;
       7 : OUT := I7
    END (*case*)
 END
```

-3.08-



SYSTEM VERIFICATION

Behavioural description

- can be executed in software
- this is called simulation
- simulation — works serially
- hardware — works in parallel

$\Longrightarrow$

SPECIAL HARDWARE DESCRIPTION LANGUAGES

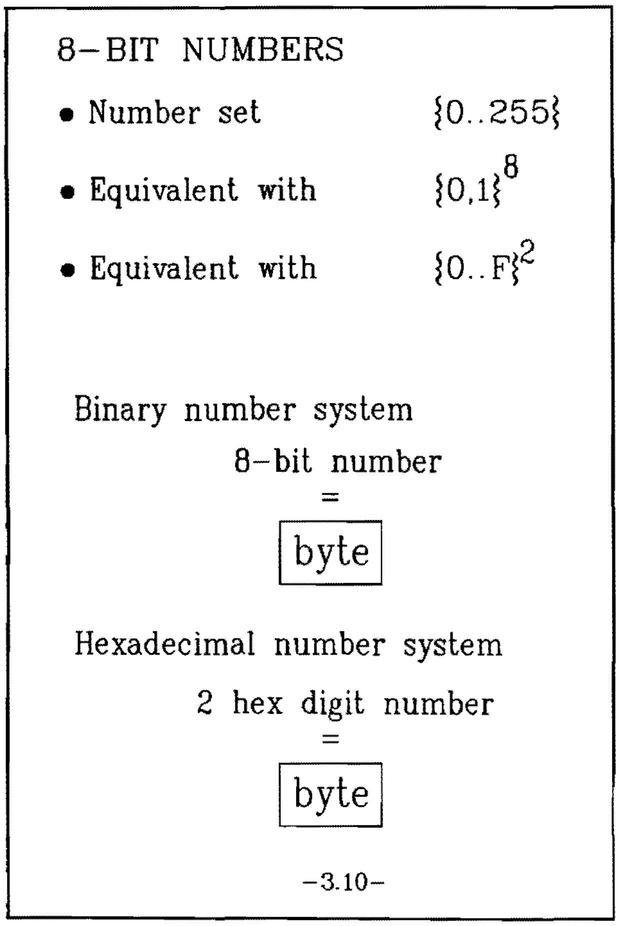
- HHDL (Silvar Lisco)
- VHDL (different companies)
- SID (Sagantec)
- IDaSS (TUE, group Digital Systems)
- etc, etc.

-3.09-

Such a multiple IF-THEN construct can be expressed more elegantly using the Pascal CASE statement. This is shown in figure 3.08. The "CASE SEL OF" construct is followed by a list of all possible values of SEL. For a given value of SEL only the statement following the corresponding label will be executed, giving the output OUT its correct value. In the multiplexer example we have encountered a new aspect of digital systems. The multiplexer performs a function to its inputs depending on the value of another input. We shall call this other input the *control* or *selection* input. The function of the multiplexer is determined by the value of this special input. We say also that the multiplexer in driven (controlled) by a control input. We have now used two words: control and selection, which play an important role in system design. We shall return to this topic later.

We have now discussed some examples of simple behavioural descriptions. Here we shall make some remarks. We have used an executable programming language to describe the behaviour of our systems.

As stated in figure 3.09, this has the extra advantage that the behavioural description can be executed (on a computer), i.e. the behavioural description is executable. Therefore, we can compute the behaviour of the digital system, simulate it. There are some disadvantages related to simulations. An important problem is that hardware implementations of have inherent parallelism, i.e. different actions are carried out simultaneously. Many programming languages are sequential, and Pascal is not an exception. That is to say that successive statements are executed sequentially. "First this and then that". To remove this and other problems, specific description languages have been constructed. With these languages we can describe and simulate parallel actions. Such languages are called *hardware description languages.* Figure 3.09 lists the names of some hardware description languages. Sometimes such a language is a part of a total system in which hardware can be described graphically and simulated. An example of such a system is IDaSS (Interactive Design and Simulation System) developed in the digital systems group of Eindhoven University of Technology. Later we shall see some applications of this system.



8-BIT NUMBERS

- Number set $\{0..255\}$
- Equivalent with $\{0,1\}^8$
- Equivalent with $\{0..F\}^2$

Binary number system

8-bit number

=

$\boxed{\text{byte}}$

Hexadecimal number system

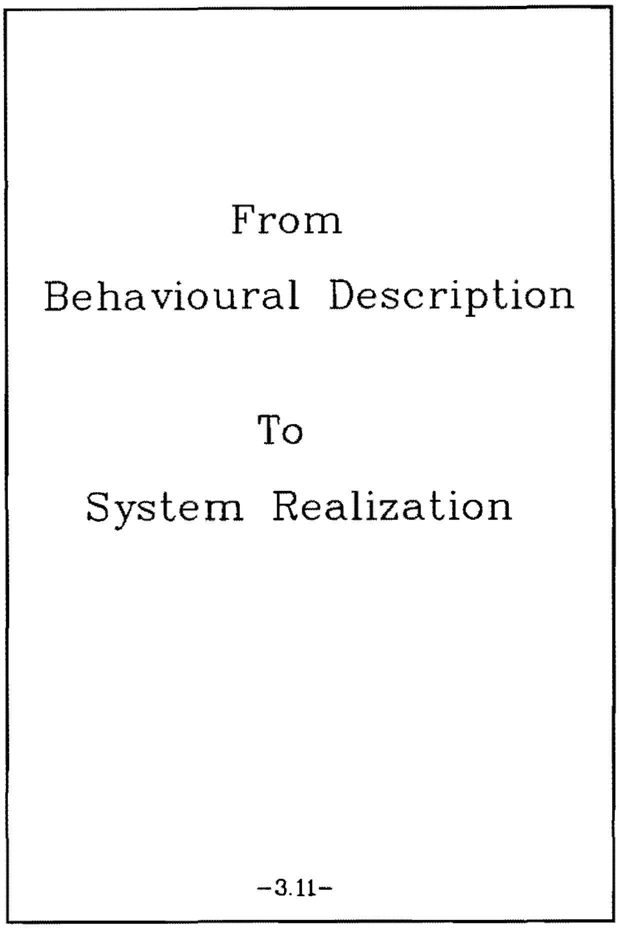
2 hex digit number

=

$\boxed{\text{byte}}$

-3.10-

Secondly, we want to make a remark about the sets of numbers used in the various examples. Why did we choose a set of numbers from 0 to 255 instead of, for example, from 0 to 99? This choice is clarified in figure 3.10. From chapter 2 we know that the chosen numeric range is identical with the range of numbers expressible in an 8-bit binary number. Thus we can simply convert our number set to an 8-bit binary number or, alternatively, to a 2-digit hexadecimal number. This choice is influenced by practice, where it is common to perform processing on 8-bit binary numbers or their multiples. A group of 8 bits is a very common unit called a *byte*. Thus we usually consider a 1-byte number, a 2-byte number, or an n-byte number. Also, as shown in figure 3.10, a byte can be represented by two hexadecimal digits.



From

Behavioural Description

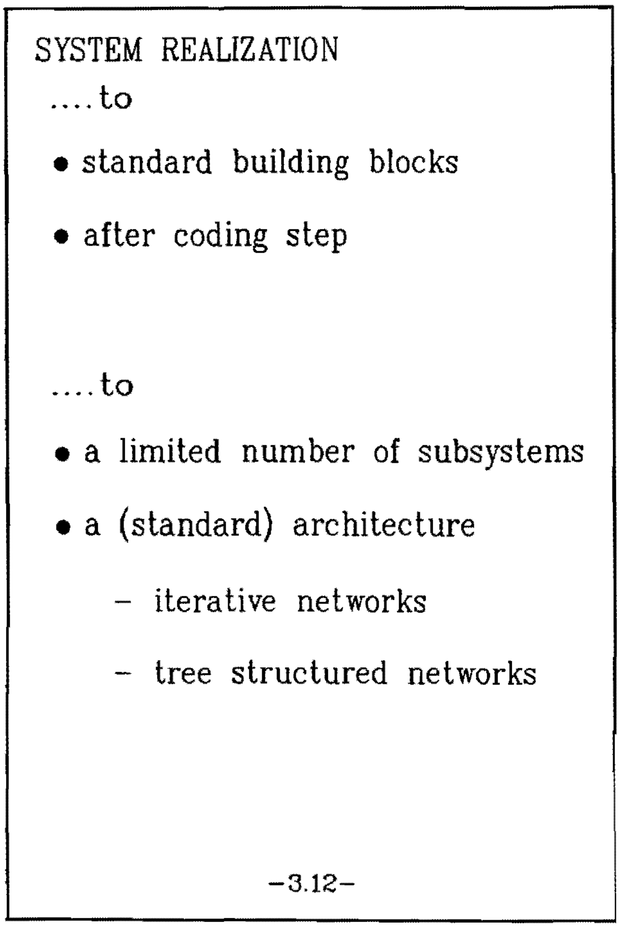
To

System Realization

-3.11-

3.2 From behavioural description to system realization

After specifying the desired system behaviour and verifying it by means of simulation we arrive at a point in the system design trajectory where we must consider a possible realization. That is to say, how can the system be built from a limited number of subsystems that are simpler to realize. Generally spoken there is no direct answer to this question. There are no known methods to find an optimal decomposition of a system into subsystems. In this decomposition lays the core of what I would like to call the "art of design".



SYSTEM REALIZATION

....to

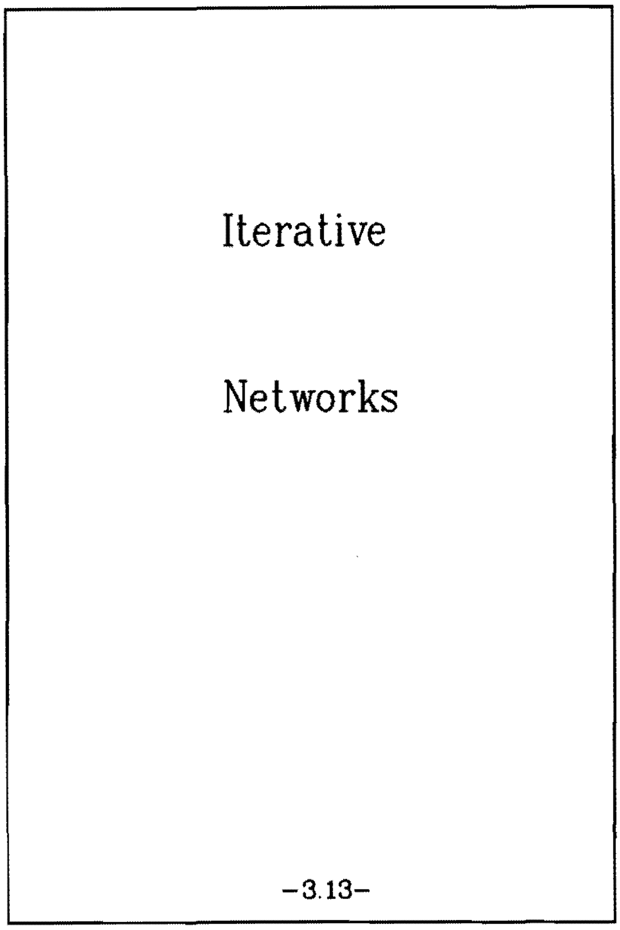
- standard building blocks
- after coding step

....to

- a limited number of subsystems
- a (standard) architecture
  - iterative networks
  - tree structured networks

−3.12−

A couple of general direction will be pointed out. Figure 3.12 first points out that we must know whether the function that we want to make is realizable using standard building blocks. To give a satisfactory answer to this question it is necessary to have an extensive knowledge of the available building blocks. To this end we shall often need to be supported by an expert (system). In other circumstances there might be programs that can automatically translate our system description into a realization that uses standard building blocks. This is called *silicon compilation.* Silicon compilation will first be realized for the simpler (sub)systems. Frequently the system behaviour is modeled using a black box model. Later we shall discuss methods that translate black box models into realizations with standard building blocks. If the (sub)system is (still) too complex, and thus these possibilities are not available, then we shall be guided by, among other things, our own creativity. We can try to find a relation with related and frequently used standard architectures. We shall discuss two of them here. First the iterative networks and then the tree structured networks.



Iterative

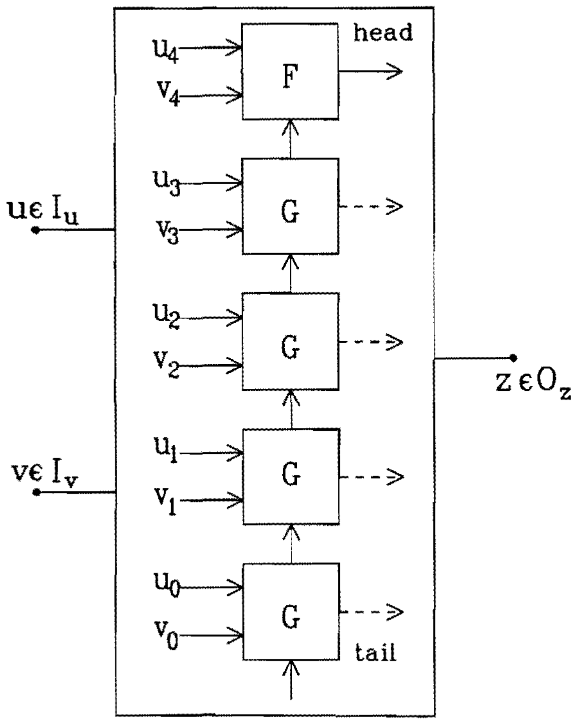
Networks

−3.13−

3.3 Iterative networks

An iterative network in principle comprises several identical building blocks that are connected in cascade. That is to say the building blocks are connected in a row. There is a first or starting building block, that passes its results to the following building block. This following block in turn passes its results to its neighbour and so on. The first building block in the row determines, in principle, the output of the system. This first building block frequently has a different function. A k-iterative network consists of k building blocks.

5-ITERATIVE NETWORK



-3.14-

Figure 3.14 shows an example of a system that is realized as a 5-iterative network, i.e. a network with 5 building blocks. We see that the inputs u and v now are, in one way or another, equally divided among the inputs of the 5 building blocks. Furthermore, we see clearly that there is a block on the head of the list that determines the output value of the system. This building block is labelled F. Furthermore, this figure clearly shows that the cascade elements are connected in a row.

K-ITERATIVE NETWORK

REQUIREMENTS

- Let I be the value set of an input
- There must exist a value set I' such that there is a one-to-one mapping of I to $(I')^k$
- For each element of I there is a single k-tuple $(I')^k$ and vice versa

EXAMPLE - 5-ITERATIVE NETWORK

- There is a: I'_u and I'_v
- with: $u_4.u_3.u_2.u_1.u_0 \in I'_u$
- and: $v_4.v_3.v_2.v_1.v_0 \in I'_v$
- such that

$$u \in I_u \longleftrightarrow (u_4.u_3.u_2.u_1.u_0) \in (I'_u)^5$$
$$v \in I_v \longleftrightarrow (v_4.v_3.v_2.v_1.v_0) \in (I'_v)^5$$

-3.15-

Are there requirements that a system, or its input values, must fulfil to make it possible to realize it by a k-iterative network? Figure 3.15 gives the answer to this question. It appears that one should be able to map the value sets of each system input neatly to the individual value sets of the inputs of the subsystems that form the iterative network. Formally stated, one should be able to write the value set of each system input as a k-tuple of another value set and vice versa. For our iterative network of figure 3.14 it must be valid that each u and v of the set of input values can be mapped one-to-one to the 5-tuple (u4, u3, u2, u1, u0) for u and the same for v.

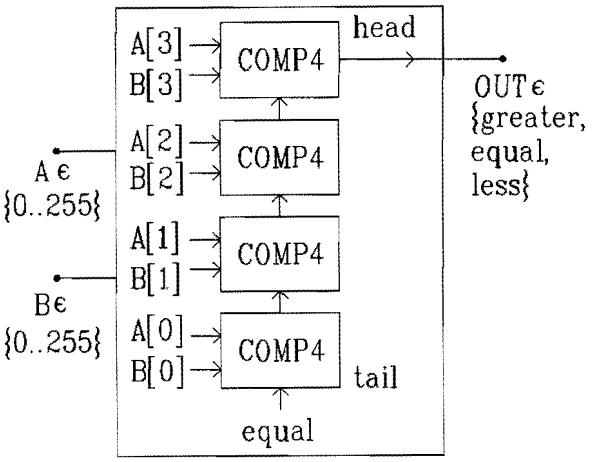
EXAMPLES - MAPPING

- Every element in $\{0..999\}$ can be mapped one-to-one to a triple in $\{0..9\}^3$
- The set $\{0..255\}$ <u>cannot</u> be mapped one-to-one to the set $\{0..9\}^3$
- The set $\{0..255\}$ can be mapped one-to-one to
  - $\{0..15\}^2$ or $\{0..F\}^2$ hex. number system
  - $\{0..3\}^4$ quaternary number system
  - $\{0,1\}^8$ binary number system
- The set $\{$Sunday..Saturday$\}$ <u>cannot be meaningfully</u> mapped to a set of n-tuples

-3.16-

EXAMPLE - COMPARATOR

- Inputs $A,B \in \{0..255\}$
- One-to-one mapping to $\{0..3\}^4$
- 4-iterative network



$A[i],B[i] \in \{0..3\}$

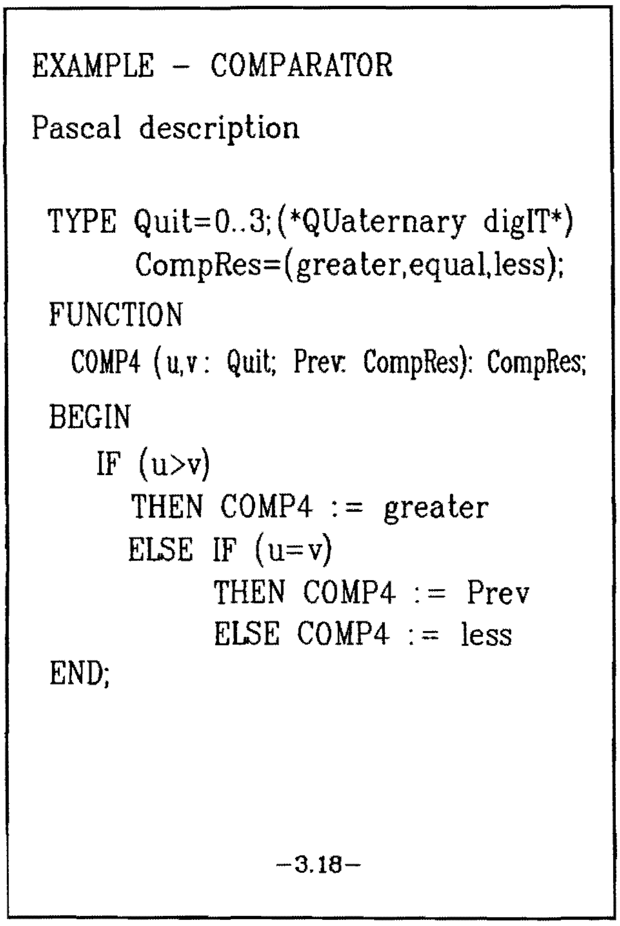
$$A = A[3]*4^3+A[2]*4^2+A[1]*4^1+A[0]*4^0$$

-3.17-

Figure 3.16 clarifies by a few examples what this rule means. First we see that each element of the set of integer numbers for 0..999 maps one-to-one to a triple with digit elements from 0 to 9. This is precisely the way that we write the values of these numbers in the decimal system.

Example 2 shows that the set of the numbers from 0..255 cannot be mapped one-to-one to triples of the elements 0..9. The triples 259 and 715 do not belong to the value set of the input. Example 3 shows that the same value set can be mapped one-to-one to duples with elements equal to hexadecimal digits. We also can represent the same value set by quadruples with elements from 0 to 3. Finally we also can present the value set by binary 8-tuples. A 2-iterative network, each of its subsystems accepting hexadecimal digits as input, can be used to realize a system with such a value set. On the other hand we can realize the same system by an 8-iterative network where each subsystem takes binary values as inputs. As a fourth example we consider the days of the week. For this set of elements it is not clear how it can be represented by a set of k-tuples. Therefore, we must think before realizing "Is X a weekday?" as a k-iterative circuit. The question is then whether the iterative circuit is the best architecture here.

As an example of an iterative circuit realization we shall discuss the realization of the modified comparator shown in figure 3.05. The main schematic is shown in figure 3.17. Here we see that the inputs, with a value range from 0 to 255, are represented by quadruples of elements with a value range from 0 to 3. We search for a realization of a 4-iterative network. In figure 3.17 we have shown that we shall attempt to realize the comparator with the help of 4 identical sub-circuits called COMP4. We see, because of this scheme, that the start circuit has a redundant input. We shall set this input to the value "equal". We are then considering a requirement or a *pre-condition*. Furthermore, we notice that the elements of the quadruples A[i] are written as elements of a row or an array. We have done so to be able to construct a Pascal description of the system. In this Pascal description the behaviour of the individual COMP4 subsystems must be defined, and the way these subsystems work together also must be defined.



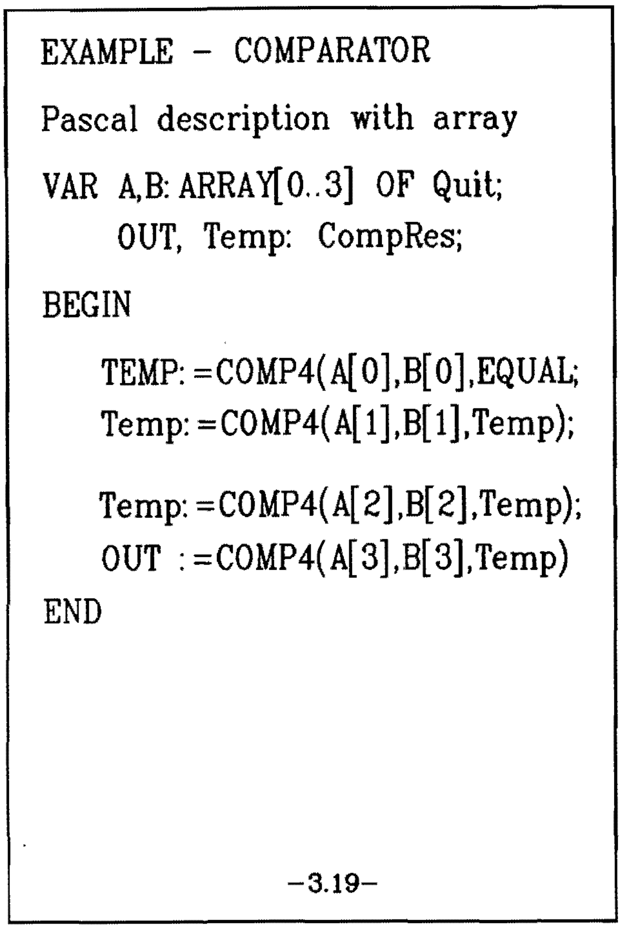
EXAMPLE - COMPARATOR

Pascal description

```
 TYPE Quit=0..3;(*QUaternary digIT*)
        CompRes=(greater,equal,less);
 FUNCTION
  COMP4 (u,v: Quit; Prev: CompRes): CompRes;
 BEGIN
    IF (u>v)
       THEN COMP4 := greater
       ELSE IF (u=v)
              THEN COMP4 := Prev
              ELSE COMP4 := less
 END;
```

-3.18-

In figure 3.18 we first find the behavioural description of the subsystem COMP4. We have defined this behaviour as a Pascal function. Such a function can be seen as a stand alone piece of description that we can refer to later. In figure 3.18 we have first defined the type of input and the type of output of our system COMP4. If we compare the description of COMP4 with the previously given description of our larger comparator (see figure 3.05) we notice that both descriptions are practically identical. There is one exception, however. If the inputs u and v are equal then the comparator result will not be "equal". However it will be equal to the result of the previous comparator in the iterative network. This is given by the variable Prev. The behaviour of the total comparator is explained as follows: The first COMP4 (in the iterative network) compares the most significant part of the input numbers. If they are equal then the least significant part of the input numbers is compared in the following COMP4 in the row etc. If all parts of both numbers are identical then the last COMP4 will ultimately pass "equal" to the above.



EXAMPLE - COMPARATOR

Pascal description with array

```
VAR A,B: ARRAY[0..3] OF Quit;
     OUT, Temp: CompRes;
BEGIN
     TEMP: =COMP4(A[0],B[0],EQUAL;
     Temp: =COMP4(A[1],B[1],Temp);

     Temp: =COMP4(A[2],B[2],Temp);
     OUT :=COMP4(A[3],B[3],Temp)
END
```

-3.19-

Figure 3.19 shows the behavioural description of the whole system described in Pascal. Notice that we define the system variables A and B as quadruples that are described as arrays of elements. Furthermore, we still need a variable to connect the COMP4 elements. This variable is called Temp. In this description, which is exclusively correct for a sequential language such as Pascal, we can clearly derive the cascade circuit. The tail COMP4 determines the result that is to be used by the following COMP4 in the cascade circuit, which internally passes a result to its neighbour. This goes on until we arrive at the component that can determine the output of the total system. We shall come back to this behaviour later when we discuss the timing aspects of iterative networks.
We also can make use of the iterative language constructs in Pascal to describe an iterative circuit.

EXAMPLE - COMPARATOR

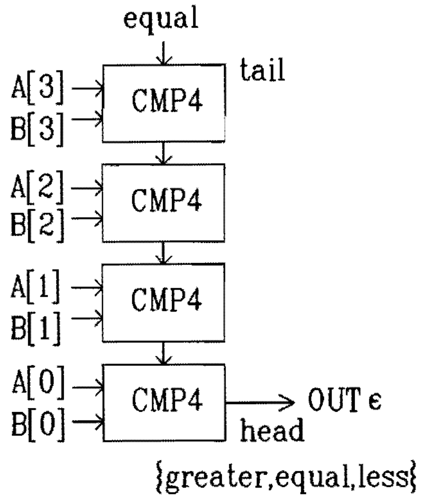
Pascal description
with array and repetition

```
 VAR A,B :  ARRAY[0..3] OF Quit;
            OUT, Temp: CompRes;
            I: Integer;
  BEGIN
    Temp := equal;
    FOR I := 0 TO 3
    Temp := COMP4(A[I],B[I],Temp);
    OUT  := Temp
  END
```

-3.20-

EXAMPLE - COMPARATOR

Alternative implementation



$A[i],B[i] \in \{0..3\}$

$A = A[3]*4^3+A[2]*4^2+A[1]*4^1+A[0]*4^0$

$B = B[3]*4^3+B[2]*4^2+B[1]*4^1+B[0]*4^0$

-3.21-

Figure 3.20 gives an example of the comparator. Instead of writing four almost identical statements we also can make use of a "FOR" statement. This is done in figure 3.20. Furthermore, such a description is a good start for the realization of the system as a sequential circuit.
We shall return to this point later.

Problem 3.4
When realizing a comparator as an iterative circuit it is also possible to compare the most significant part of both numbers in the tail circuit, and the least significant part in the head circuit. Figure 3.21 shows a block diagram of such a comparator. Give a Pascal description of the behaviour of the components COMP4 and the whole behaviour of the total comparator.

SUB-OUTPUTS

- Implementation of
  k-iterative network
  with
  k sub-outputs
- Condition:
  Every output value 0
  can be mapped one-to-one to
  a k-tuple in $(0')^k$

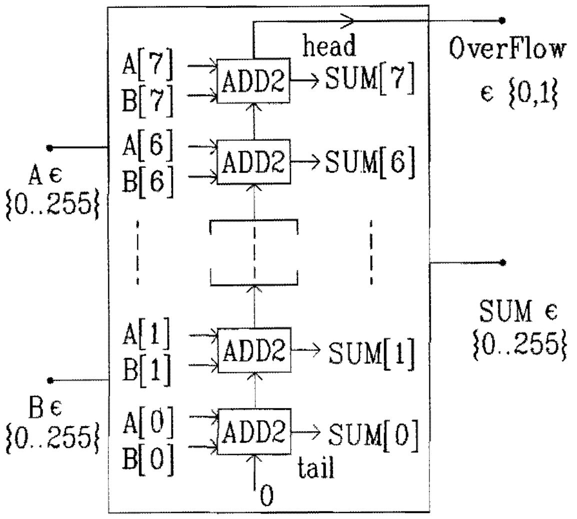
EXAMPLE

The output values
{greater,equal,less}
<u>cannot be meaningfully</u> mapped
to a set of k-tuples

-3.22-

EXAMPLE - ADDER

- A,B,SUM ∈ {0.255}
- Overflow∈ {0,1}
- {0..255} is mapped one-to-one to $\{0,1\}^8$
- 8-iterative network:

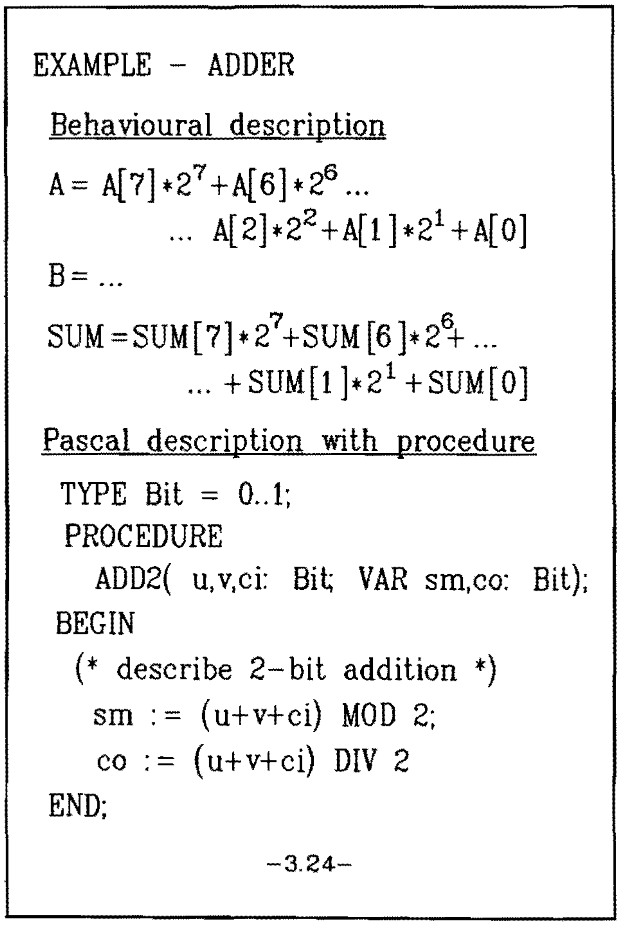


A[i],B[i],SUM[i] ∈ {0,1}

-3.23-

The dotted lines leaving each element G, in figure 3.14, show that it is also possible to determine the output values of all elements simultaneously. Here we are considering a k-iterative network with k partial outputs (see figure 3.22). This is only possible for outputs with value sets that can be mapped one-to-one to a k-tuple of elements of another set. Figure 3.22 shows that the output value set of our comparator ("greater", "equal" and "less") could not be meaningfully mapped to k-tuples. Therefore, the comparator cannot be realized as a k-iterative network with k sub-outputs.

The adder is a circuit that can be realized as an iterative circuit with partial outputs. The adder, previously described in figure 3.04, can be realized as an 8-iterative network. Figure 3.23 shows the structure of the iterative network. Notice that inputs A and B and output SUM have a one-to-one mapping on a binary 8-tuple. Each sub-circuit ADD2 adds 2 bits of both operands and produces one bit of the SUM result. The *carry* is passed to the following circuit in the row, beginning by the tail circuit and ending as overflow at the head circuit.



EXAMPLE - ADDER

Behavioural description

$A= A[7]*2^7+A[6]*2^6 \ldots$

$\ldots A[2]*2^2+A[1]*2^1+A[0]$

$B= \ldots$

$SUM=SUM[7]*2^7+SUM[6]*2^6+ \ldots$

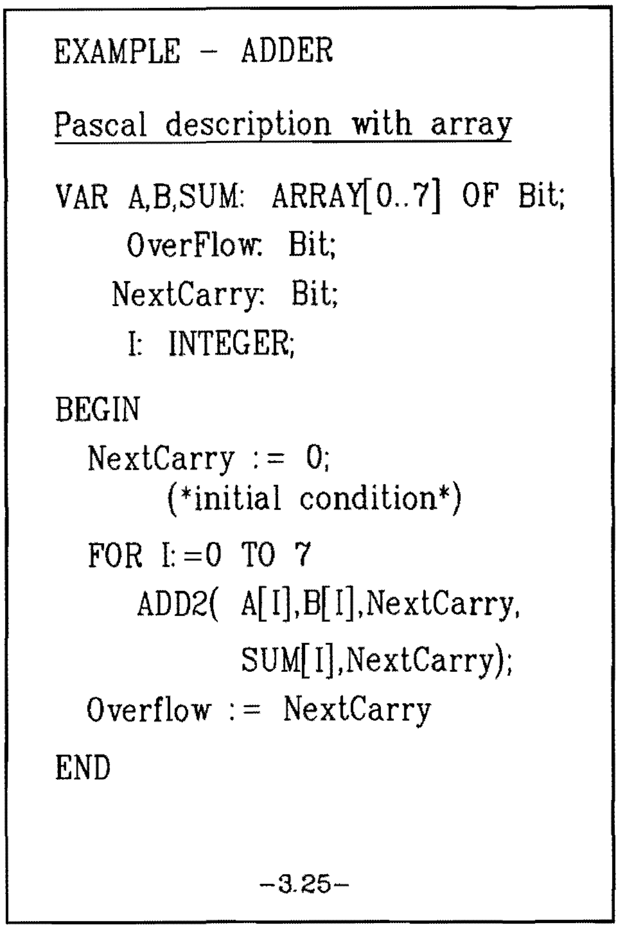
$\ldots +SUM[1]*2^1+SUM[0]$

Pascal description with procedure

```
 TYPE Bit = 0..1;
 PROCEDURE
   ADD2( u,v,ci: Bit; VAR sm,co: Bit);
BEGIN
  (* describe 2-bit addition *)
   sm := (u+v+ci) MOD 2;
   co := (u+v+ci) DIV 2
END;
```

-3.24-

The behavioural description of the circuit ADD2 is given in figure 3.24. Because we now have more than one output per circuit we cannot make use of Pascal functions. Instead we must use a procedure. Therefore the VAR parameters sm and co are both outputs from the circuit ADD2. The behaviour of ADD2 is described by a 2-bit addition statements. To describe this we can make use of the modulo and integer division operators.



EXAMPLE - ADDER

Pascal description with array

```
VAR A,B,SUM: ARRAY[0..7] OF Bit;
     OverFlow: Bit;
    NextCarry: Bit;
     I: INTEGER;
BEGIN
  NextCarry := 0;
       (*initial condition*)
  FOR I:=0 TO 7
     ADD2( A[I],B[I],NextCarry,
           SUM[I],NextCarry);
  Overflow := NextCarry
END
```

-3.25-

Problem 3.5

Verify that the description of figure 3.24 is related to the previously mentioned statements of adding two bits (fig 2.40).

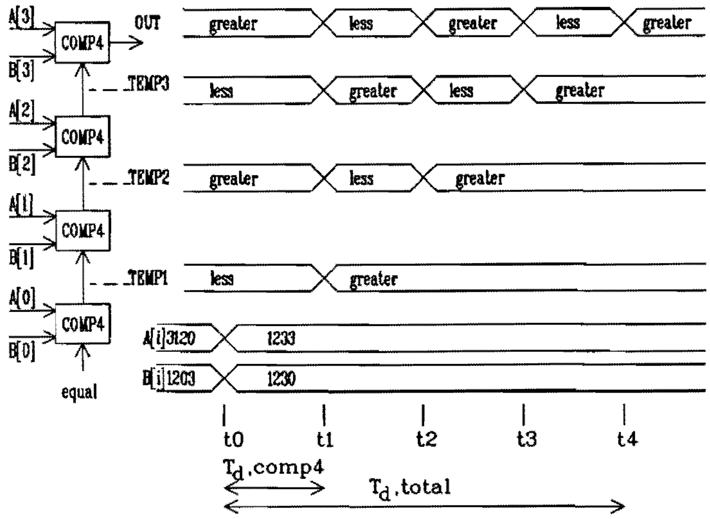
The behaviour of an 8-bit adder can be described iteratively as shown in figure 3.25. There the variable NextCarry is used for saving the carry of the 2-bit adder. In the "FOR-loop" the 8 bits of both inputs are successively added to each other and the sum and the new NextCarry are determined.

Timing Aspects in

Iterative Networks

-3.26-

TIMING - 4-ITERATIVE COMPARATOR



-3.27-

In the given behavioural description of the adder timing considerations arise. Thus far we have not considered the time necessary to execute a combinational function. The execution of a combinational logic function using an electronic circuit costs a certain amount of time. This time might be very short, in the order of nano-seconds for a basic operation, but can lead to too slow realization when used in iterative networks.

Figure 3.27 schematically shows the behaviour in time of a 4-iterative comparator. Such a diagram is called a timing diagram. At the bottom of timing diagram the values of the inputs A and B are shown symbolically. At time t0 the values of the quadruples 3120 and 1203 change to the quadruples 1233 and 1230, for A and B respectively. All four comparators need some time to process these input values. We see that (initially) at time t0 there is no change in the outputs (TEMP1 to TEMP3 and OUT) of these comparators.

After a certain delay time T_d, i.e., at time t1, the values of these outputs change. These value changes, at that time point, are only a result of the change of the input values A[I] and B[I]. Indeed, the inputs TEMP1 to TEMP3 remain unchanged until time t1.

Now these inputs take on new values and after a certain delay, i.e., at time t2, the outputs TEMP2 to TEMP3 and OUT change their values again. Both head comparators in the iterative network are once more confronted by a new change of an input value. After a new delay time, i.e. at time t3, they will adjust their outputs TEMP3 and OUT. Finally, after another delay time, i.e., at time t4, the head comparator will produce its output value. We see that a change of the inputs ripples through the system from tail to head.

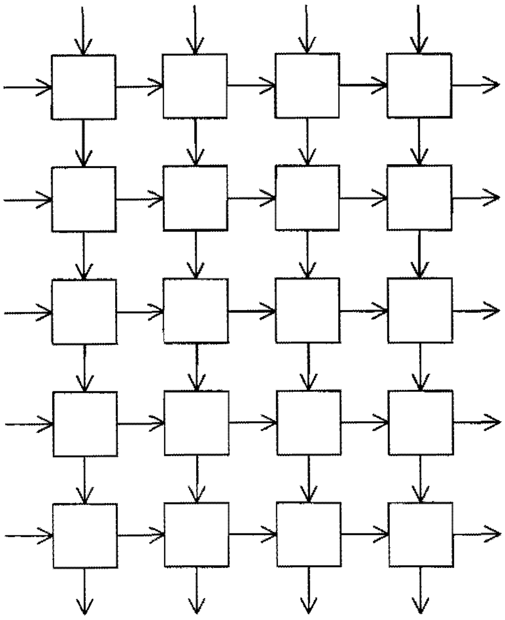
RIPPLE-THROUGH BEHAVIOUR

- Total delay =
  k * (delay of single module)

- Lookahead networks

-3.28-

MULTI-DIMENSIONAL
ITERATIVE NETWORKS



-3.29-

We are considering *ripple-through* behaviour (see figure 3.28). The total delay time of a k-iterative network is equal to k times the delay time of a network module. This can be a considerable disadvantage especially for iterative networks with several cascaded elements. The delay time becomes long and therefore the system possibly cannot execute its task with the desired speed. The disadvantage of iterative networks can be compensated by using so-called look-ahead-networks. The characteristics of such networks lie beyond the scope of this course.

At the beginning of this paragraph we defined an iterative circuit, very strictly, as a cascade of elements with a "tail" and a "head" element. Each element processes a part of the input and passes information to the following element in the chain. We can generalize this idea of iterative networks by allowing information exchange in the other direction, from head to tail. There are known circuit realizations where this is applied. The following step allows the use of multi-dimensional iterative circuits.

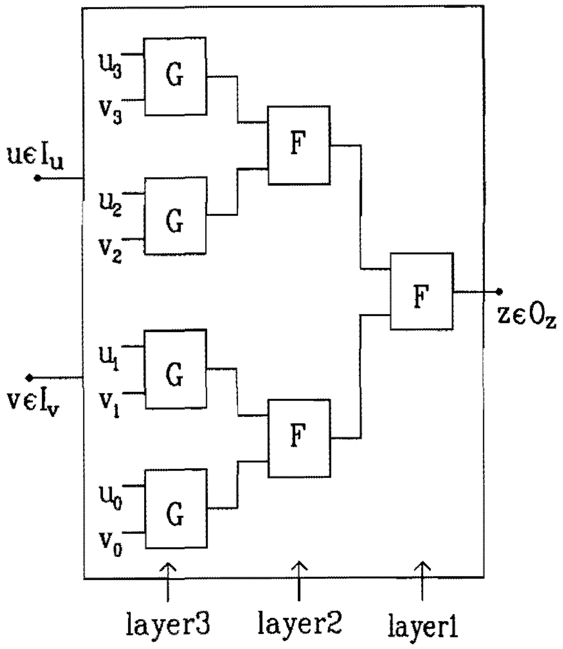
Figure 3.29 shows an example of this. Now each circuit in the iterative network has two following circuits where it can deposit information. Naturally, the following step is to allow a bidirectional (in two directions) information stream. The treatment of this type of iterative network is beyond the scope of this course.

Tree Structured

Networks

-3.30-

BINARY TREE STRUCTURE



Depth of the tree =
number of circuit layers

-3.31-

3.4 Tree structured networks

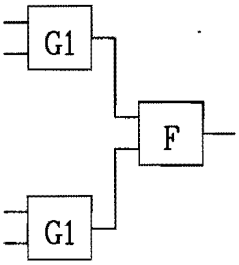
A tree structured network is a network of circuits, where the system output is connected to a circuit and this circuit has connections (branches) to other circuits. Each of these other circuits also has connections to a following layer of circuits etc. A general tree structure can be irregular and unbalanced. In this section we shall limit ourselves to networks with a binary tree structure.

Figure 3.31 shows a schematic of a network with a binary tree structure. We see that the output of the system is connected to a circuit, F, on layer 1. Both inputs of this circuit are connected to two identical F circuits on layer 2. In turn the two inputs of each of these circuits are connected to identical G circuits on layer 3. Here the circuits (on layer 3) do not branch any further but are connected to the system inputs. The depth of a binary tree is equal to the number of circuit layers we have. In figure 3.31 we have a depth of 3 layers.

SYNTHESIS OF TREE STRUCTURE

The synthesis of a tree structure is a recursive process:

- Make a <u>two-layer</u> tree structure



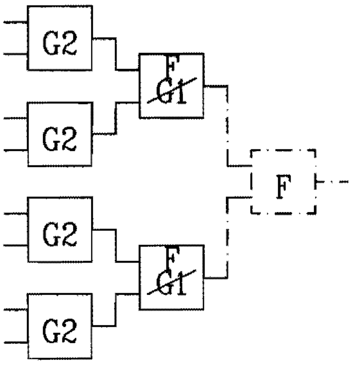
-3.32-

The synthesis of a tree structure is a recursive process. That is to say: we do not immediately realize our system as a binary tree with a depth of n layers, but we do that step by step. The recursive process is illustrated in figures 3.32 to 3.34. The first step is shown in figure 3.32. This step develops a two-layer tree structure.

SYNTHESIS OF TREE STRUCTURE

The synthesis of a tree structure is a recursive process:

- Make a <u>two-layer</u> tree structure
- Realize each of the resulting circuits G1 as a <u>two-layer</u> tree structure



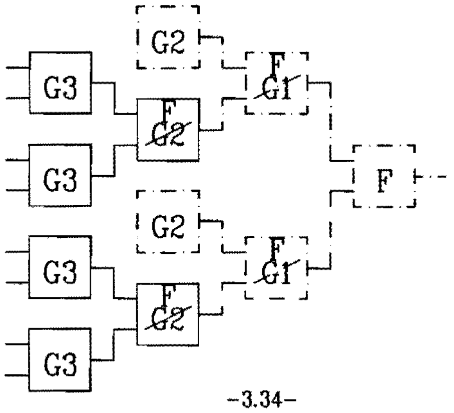
-3.33-

Subsequently we can attempt to realize each resulting G1 circuit as a new 2-layer tree structure. Herewith each G1 circuit is replaced by the F circuit (see figure 3.33).

SYNTHESIS OF TREE STRUCTURE

The synthesis of a tree structure is a recursive process:

- Make a two-layer tree structure
- Realize each of the resulting circuits G1 as a two-layer tree structure
- Realize a next layer Gi as a two-layer tree structure: this adds a next layer to the tree depth



-3.34-

TWO-LAYER BINARY TREE

Definition

- I is a value set of a system input
- I' is a value set with a one-to-one mapping of I to $(I')^2$
- For each $i \in I$ there is an $i' \in (I')$ and vice versa

Examples

- $\{0..255\}$ can be mapped one-to-one to $\{0..15\}^2$ or $\{0..F\}^2$
- $\{0..7\}^4$ can be mapped one-to-one to $(\{0..7\}^2)^2$ (split a 4-digit number in two groups of 2 digits)

-3.35-

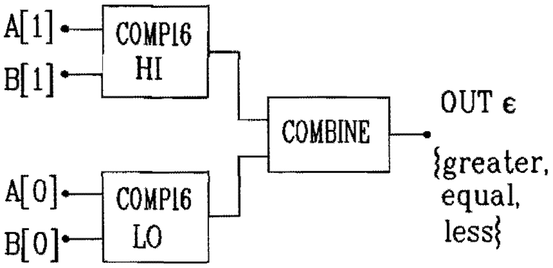
After these two steps we have realized the system as a 3-layer tree structure. The realization of a following Gi as a 2-layer tree structure adds a following layer to the depth of the tree. This is shown for the two G2 circuits in figure 3.34. As we make the same step for both of the other G2 circuits we have a complete 4-layered tree structure.

We see that the development of the tree structure depends on the possibility to realize our system as a 2-layer binary tree. A condition for the existence of binary tree realization is shown in figure 3.35. This condition boils down to the condition that the value set of each input of the system must have a one-to-one mapping to a duple. A binary tree will generally spoken not always be effective, however.

Figure 3.35 gives a few examples. First the set of integers from 0 to 255 can be mapped to duples with elements from the set 0 to 15 or the hexadecimal digit symbols. Another example shows the mapping of quadruples with elements from 0 to 7 to duples, which are in turn composed of duples. We split a 4-digit number into two groups of two digits.

EXAMPLE - COMPARATOR

Inputs A,B ε {0..255}
One-to-one mapping to $\{0..15\}^2$
(see fig. 3.05)



```
TYPE Hit = 0..15; (* HexdigIT *)
CompRes = (greater,equal,less)
```

-3.36-

EXAMPLE - COMPARATOR

Behavioural description: functions

```
FUNCTION
COMP16(u,v: Hit): CompRes;
BEGIN
  IF (u>v)
    THEN COMP16 := greater
    ELSE IF (u=v)
           THEN COMP16 := equal
           ELSE COMP16 := less
END;

FUNCTION
COMBINE(Hi,Lo: CompRes): CompRes;
BEGIN
  IF (Hi = equal)
    THEN COMBINE := Lo
    ELSE COMBINE := Hi
END;
```

-3.37-

As an example of realization with a binary tree structure we shall discuss the comparator. Figure 3.36 shows a schematic of a 2-layer tree-structured decomposition. The range of both inputs A and B is mapped to 2 hexadecimal digits. The two most significant digits are compared in the upper COMP16 and the least significant digits are compared in the lower COMP16. The results of both comparators are combined in the circuit COMBINE. We see that our system is built from two different circuits, thus we also must deliver two behavioural descriptions.

Figure 3.37 shows both behavioural descriptions as two functions. Notice that the function COMP16 describes the standard behaviour of a comparator. If the value of the variable u is greater than the value of the variable v, then the result of the comparator is "greater". The result is "equal" if u equals v, and otherwise the result is "less". The function COMBINE must now combine the results of both comparators. This is a very simple function as shown in the behavioural description. The comparison of both most significant hexadecimal digits determines whether A > B unless both digits are equal. Then the comparison of the least significant digits determines the result of the comparison. This is precisely the description of the behaviour of the COMBINE function.

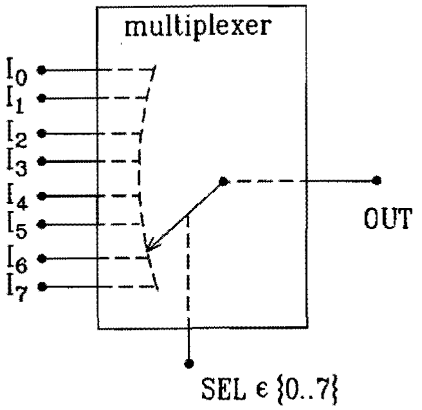
EXAMPLE - COMPARATOR

Behavioural description:
Use of functions

```
VAR A,B: ARRAY [0..1] OF Hit;
    OUT : CompRes;
BEGIN
  OUT: =
        COMBINE (COMP16(A[1],B[1]),
                 COMP16(A[0],B[0]));
END
```

-3.38-

EXAMPLE - MULTIPLEXER



- SEL is a control input (selection input)
- SEL can be split into two control inputs:
  - from SEL ∈ {0..7} to
  - duple (SELa,SELb) ∈ {0,1}*{0..3}

-3.39-

Figure 3.38 shows the behavioural description of the total system, making use of the functions COMBINE and COMP16. This description is self-explanatory.

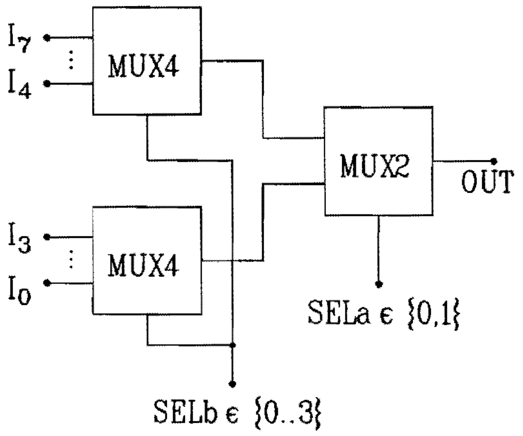
Problem 3.6

Show that the COMP16 module can be realized as a 2-layer tree-structured network. Give a Pascal description of your solution.
How does the tree of the total system look?
How many layers does the tree have?

We have seen earlier that the multiplexer takes a distinct place among the combinational circuits. The multiplexer's special characteristic is that it has a particular input, the control or select input, which determines the function of the multiplexer. Figure 3.39 shows this again. We can achieve a binary tree realization by dividing the control function into a duple (instead of mapping the inputs into duples). So we get the elements SELa and SELb (see figure 3.39).

EXAMPLE - MULTIPLEXER

- Use SELa to select one of the groups $(I_0..I_3)$ or $(I_4..I_7)$
- Use SELb to select one input from the selected group

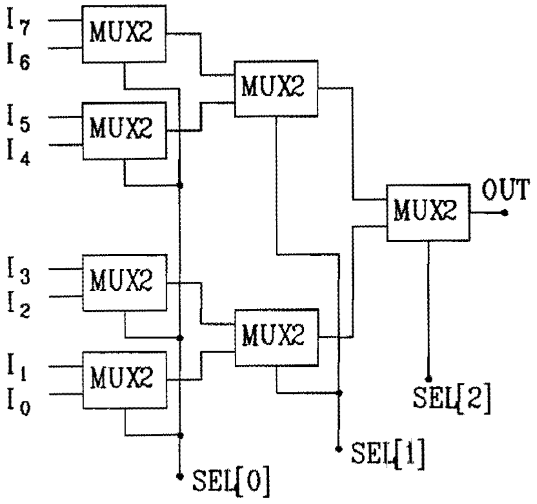


-3.40-

EXAMPLE - MULTIPLEXER

Realization with further splitting of control input

- SELb is split into (SELx,SELy) $\in \{0,1\}^2$
- This results in a 3-layer tree structure:



-3.41-

Figure 3.40 shows the consequence of this on the binary tree realization of the multiplexer. With selection input SELa we select an input from the groups I_0 to I_3 or I_4 to I_7. With selection line SELb and the two multiplexers in layer 2 we chose one of the four possible inputs from both groups. Each multiplexer in layer 2 (i.e., the MUX4 circuits) can be again realized as a 2-layer tree structure.

Finally we arrive at a schematic such as the one shown in figure 3.41. We have now mapped the values of the selection line to a binary triple. Each element of the triple is used at its own layer in the binary tree to select between those two possible inputs. We see that the multiplexer is built from MUX2 circuits, i.e., 2-input multiplexers.

Problem 3.7

In figure 3.04 of the survey the behaviour of an adder is shown.

a. *Specify a 3-layer tree-structure realization for this adder.*

b. *What is paritcular about layer 1 and 2?*

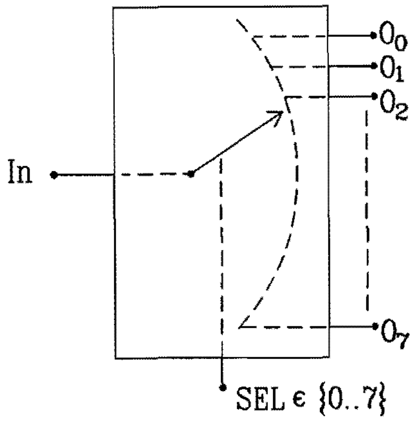
c. *Give a behavioural description of the different layers.*

EXAMPLE - MULTIPLEXER

```
TYPE BIT = 0..1; (*Binary digIT*)

FUNCTION
  MUX2( Ia,Ib: AnyType; Selx: Bit): AnyType;
  BEGIN
    IF (Selx=1)
       THEN MUX2 := Ia
       ELSE MUX2 := Ib;
  END;
```

-3.42-

EXAMPLE - DEMULTIPLEXER



- $I_n, O_0, O_1 \dots O_7 \in W\{..\}$
- SEL is a selection input
- SEL selects a certain output which will continuously copy the input value
- The other outputs have a default value from W

-3.43-

Figure 3.42 shows a description of the function of these 2-input multiplexers. The behavioural description is now very simple. We only have a few alternatives. It is either Ia or Ib that is passed to the output. There are no other possibilities. This is shown in figure 4.42.

Problem 3.8
Give, making use of the behavioural description of MUX2 shown in figure 3.42, a description of an 8-way multiplexer, and show the structure of this multiplexer.

Problem 3.9
Consider the multiplexer in fig 3.39 of the survey.

a. Is a k-iterative realization possible, using 2-input multiplexers (MUX2)?

b. If so:
- What is the value of k?
- Show a circuit diagram.
- What difference is there in comparison with standard iterative circuits?

We have now seen how we can get another type of control structure by mapping the control or selection input to an n-tuple. We can get yet another type of tree structure when realizing a demultiplexer. A demultiplexer is the inverse circuit of a multiplexer. We have now a system with one input In and several outputs. Figure 3.43 shows these output O_0 to O_7, 8 outputs in total. Besides that, the demultiplexer has a control input. This control input decides which output is connected to the input. The unselected outputs have a default value from the set of possible output values.

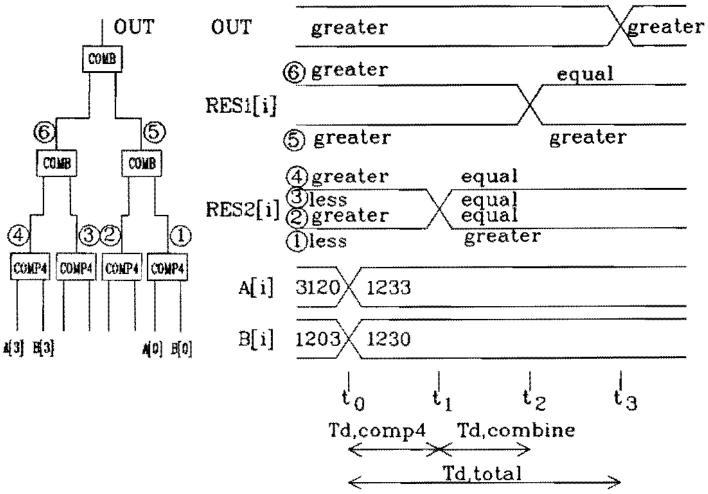
Problem 3.10
Give a Pascal behavioural description of the 1 to 8 demultiplexer shown in figure 3.43. We do not consider the value of the unselected outputs.

Problem 3.11
Give a tree-structured realization of the 1 to 8 demultiplexer. What do you notice when comparing this tree structure with the multiplexer's tree structure?

Timing Aspects in Networks with a Tree Structure

-3.44-

TIMING - 3-LAYER COMPARATOR



-3.45-

As with iterative networks time also plays an important role with tree structured networks. Also here the question arises how fast a tree structured network can execute a defined combinational function. To simplify the comparison with iterative networks we shall again discuss the comparator. Figure 3.45 shows the realization of a comparator in a 3-layer tree structure. The same figure shows a timing diagram. We see that if at time t0 the input quadruples change their value; then the comparators in layer 3 (the COMP4 circuits) also will change their values. The timing diagram shows the values of these outputs (item 1 to 4). We see that after a certain delay time T_d the comparators change their values. This happens at time t1. Consequently the inputs of the COMB circuits at layer 2 changes after a certain delay time; at time t2 the outputs of these circuits will take a new value. This is given in items 5 to 6 in the timing diagram. At time t2 the inputs of layer 1 COMB circuits change, and at time t3 the value of the output OUT will take a new value.

DELAY

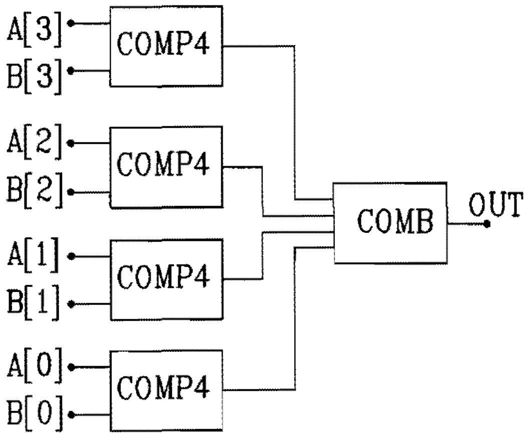
- The total delay is determined by tree depth = number of tree layers
- Number of circuits ≈ 2^{depth}

-3.46-

N-ARY TREE STRUCTURES

Comparator

- A,B ∊ {0..255} —> A,B ∊ $\{0..3\}^4$
- Consequently this is a quaternary tree structure:



A[i],B[i] ∊ {0..3}

OUT ∊ {greater,equal,less}

-3.47-

The total delay time of the binary tree circuit is determined by the depth (the number of layers) of the tree. The number of circuits in the tree is really equal to the second power of the depth of the tree. For larger systems the binary tree realization will in general be faster than an iterative circuit but it also will use more circuits than its equivalent iterative realization. The gain in speed brigs a higher cost.

Up till now we have only considered *binary* tree structures. We can similarly define ternary (3-ary), quaternary (4-ary), or in general n-ary tree structures. We shall illustrate this with two examples. In figure 3.47 the comparator is first realized as quaternary tree structure. We see that there are 4 connections from the COMB circuit on layer 1 to 4 equivalent circuits on layer 2. Each circuit on layer 2 forms a comparator that compares two numbers in the domain 0 to 3. We only have to deal with a 2-layer circuit with a shorter delay time from input to output. This is an advantage in comparison the previously discussed binary realization. The COMB circuit at layer 1 probably will be more complex, however.

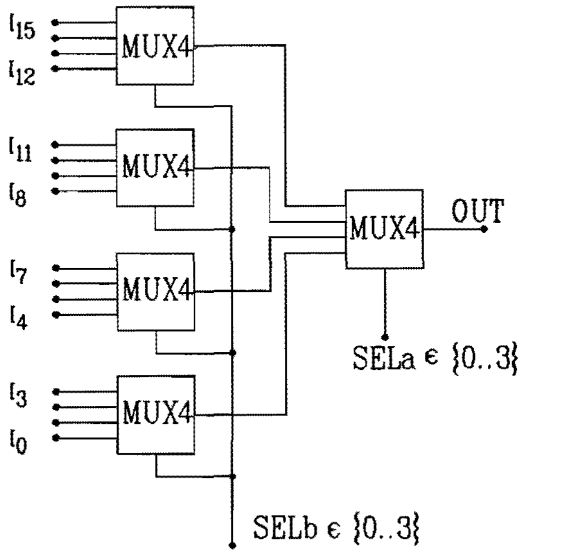
Problem 3.12

Describe the behaviour of the COMB circuit in the comparator of figure 3.47.

Compare this behavioural description with the corresponding circuit of the binary tree realization shown in figure 3.36.

N-ARY TREE STRUCTURES

- SEL ∈ {0..15} ⟶ $\{0..3\}^2$
- 2-layer quaternary tree structure



-3.48-

In figure 3.48 we find a realization of a 1 out of 16 multiplexer in a 2-layer quaternary tree structure. We have now mapped the selection input with a domain from 0 to 15 to a duple, where the elements have a range from 0 to 3.

Each of these elements controls a 1-out-of-4 multiplexer MUX4. On layer 1 of the quaternary tree structure we find again one of those multiplexers, on layer 2 we find four multiplexers. Again we see the branching to four circuits in layer 2.

3.5 Summary

This chapter dealt with the definition and construction of combinational systems. We have clearly limited ourselves to systems without memory. We have shown how we can use a formal description language such as Pascal to define the behaviour of combinational systems. It is also important to define clearly the domain and range of the system, in other words the possible values of inputs and outputs.

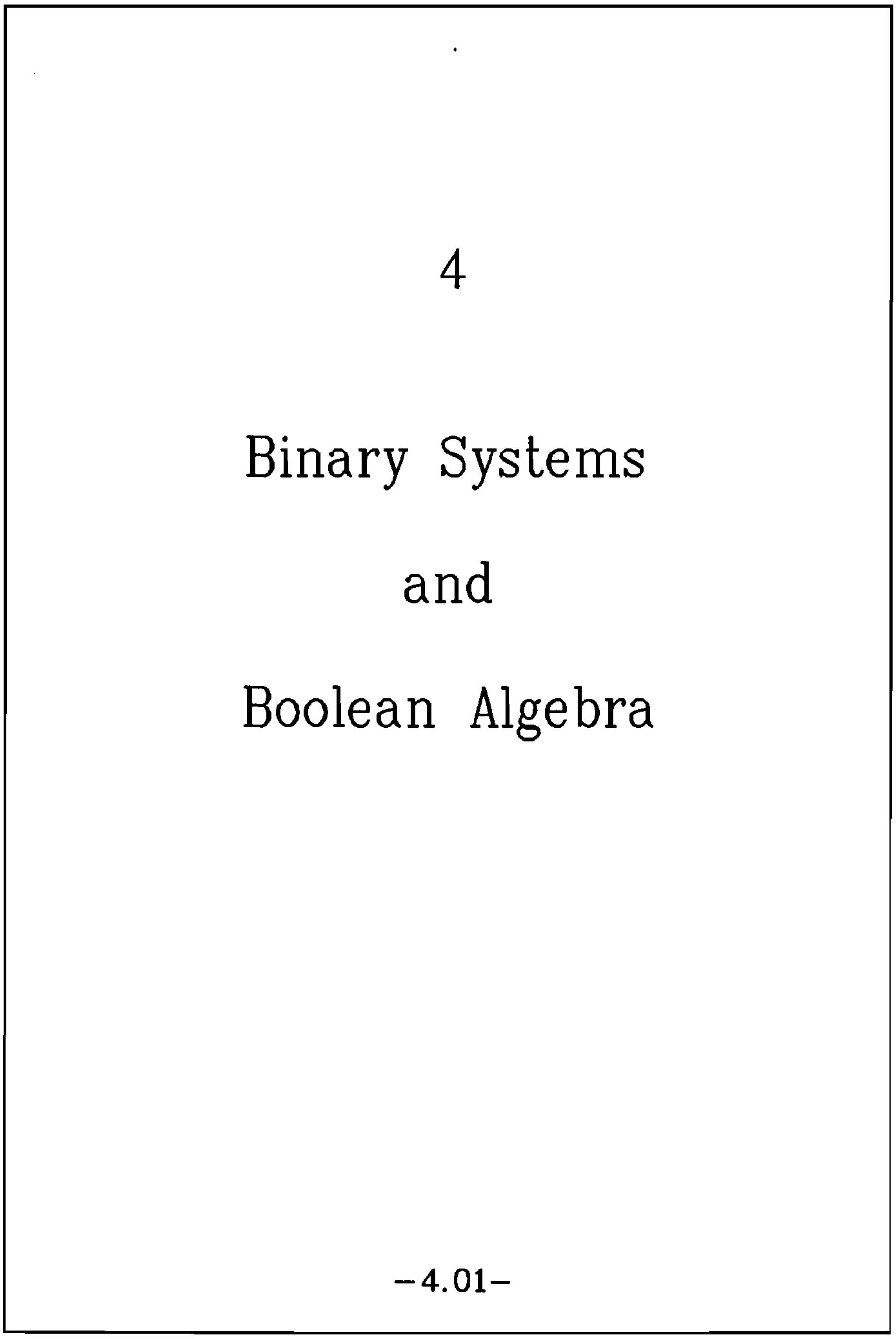
Making a good behavioural description, i.e., a good specification, is only the first step on the way towards the realization of a digital system, however. In our hierarchical structured design methodology the following step is to try to divide the system into a limited number of subsystems. We have not answered the question how to do this. In this step lies what might be called: "the art of design".

We can give a couple of directives, however, such as:

- Can we find a mapping to standard building blocks? or
- Is an association with a standard architecture a good solution?

We have discussed two standard architectures, the iterative network and the tree-structured network. In a few examples we have shown how we can develop these network structures. We have also compared the timing aspects of both network structures. We concluded that networks with a tree structure are generally faster, at the expense of a larger number of components.

Finally we notice that we totally concentrated on algorithmic descriptions. By working out several examples we see that some subsystems are really not that simple. They often require a transformation into a black box model possibly combined with a coding step. This transformation and the subsequent realization in standard building blocks is discussed in the following two chapters.



4

Binary Systems

and

Boolean Algebra

-4.01-

4 Binary systems and Boolean algebra

In the design process we eventually have to deal with small subsystems. At a certain moment a transition is made from a digital system to a binary system. This binary system should finally be realized.

We have called the transition from a digital system to a binary system *coding.* This coding step can be done separately with an explicit coding step. Actually, as we shall see, this coding step can also be implicit within the realization of our system as an iterative or tree-structured network.

CODING

digital system

⇓

coding

⇓

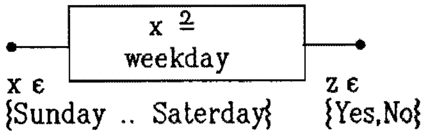
binary system

$$F : I = \{0,1\}^n \longrightarrow O = \{0,1\}^m$$

-4.02-

EXPLICIT CODING STEP

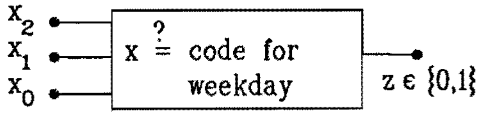
Example



Coding

Sunday	= 0 0 0	Thursday	= 1 0 0	
Monday	= 0 0 1	Friday	= 1 0 1	Yes = 1
Tuesday	= 0 1 0	Saturday	= 1 1 0	No = 0
Wednesday	= 0 1 1	Others	= 1 1 1	

Binary System



$$x_2 x_1 x_0 \in \{0,1\}^3$$

$$\begin{aligned} F = \{ & 000 \rightarrow 0, & 001 \rightarrow 1, \\ & 010 \rightarrow 1, & 011 \rightarrow 1, \\ & 100 \rightarrow 1, & 101 \rightarrow 1, \\ & 110 \rightarrow 0, & 111 \rightarrow 0 \text{ or } 1\} \end{aligned}$$

-4.03-

Figure 4.02 shows this coding step as the link between the digital system and the binary system. We shall continue to limit ourselves to combinational systems, i.e. systems without memory, where the relation between inputs and outputs is given by a function. This function is the mapping of the binary n-tuple at the input to the binary m-tuple at the output. In chapter 2 we have shown the transition from a digital system to a binary system for the function "Is X a weekday", where we explicitly made a coding step.

The related figure is repeated in figure 4.03. The chosen code for the days of the week is explicitly shown, and the resulting function of the binary system is also mentioned.

IMPLICIT CODING STEP

Example

Realization of adder:

- A, B, SUM $\in \{0..255\}$
- Type of network: 8-iterative
- A[i], B[i], SUM[i] $\in \{0,1\}$
- Function:

  ADD2 : $\{0,1\}^3 \longrightarrow \{0,1\}^2$
- Behaviour:

```
 PROCEDURE
     ADD2(u,v,ci: Bit; VAR sm,co: Bit);
   BEGIN
     sm := (u+v+ci) MOD2;
     co := (u+v+ci) DIV2;
   END;
```

-4.04-

When realizing an adder with an input range from 0 to 255 as an iterative circuit, we implicitly make a coding step. This is shown in figure 4.04. The inputs and outputs of the subsystems of the 8-iterative circuit (i.e. A[I], B[I] and SUM[I]) have a range from 0 to 1, and thus they are binary. The system ADD2, the 2-bit adder, has become a binary system by which binary input triples are mapped to binary output duples. The description of the circuit behaviour in figure 4.04 shows that the system operates on bits; binary inputs and binary outputs.

TRUTH TABLE

- Function:

  F : $\{0,1\}^n \longrightarrow \{0,1\}^m$
- A truth table comprises:

  -Each input n-tuple i $\in \{0,1\}^n$

  -Each corresponding output m-tuple o $\in \{0,1\}^m$

ADD: $\{0,1\}^3 \longrightarrow \{0,1\}^2$

ci	u	v	sm	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

-4.05-

For the type of combinational system we are discussing now, a Pascal behavioural description is no longer convenient. A specification similar to the one given for the function "Is X a weekday" is more useful. In that case we have simply written down an output value for each input tuple. We call such a table a *truth table*. When we have a mapping from an n-tuple to an m-tuple, the truth table can be constructed by writing down the input n-tuple for each output m-tuple. This is shown, as an example, for the ADD2 circuit of our 8-iterative adder. Check for yourself if this table is consistent with the circuit's behavioural description shown in figure 4.04.

BINARY SYSTEM

- Function:

  $F : \{0,1\}^n \rightarrow \{0,1\}^m$
- Rewrite F uniquely:

  -as an m-tuple of functions

  -each function defines a single binary output

  $-F = (f_1, f_2, \dots f_m)$

  -with $f_i: \{0,1\}^n \rightarrow \{0,1\}, 1 \leq i \leq m$
- The following must hold:

  $F(x) = (f_1(x), f_2(x), \dots f_m(x))$ with $x \in \{0,1\}^n$

SWITCHING FUNCTION

- A binary function is called a switching function
- $f_i: \{0,1\}^n \rightarrow \{0,1\}$

  $f_i(x_{n-1}, x_{n-2} \dots x_2, x_1, x_0)$

  = a switching function of n variables

-4.06-

A binary system is a system that realizes a mapping from an input binary n-tuple to an output binary m-tuple (see figure 4.06). On the other hand we can also interpret the function F as an m-tuple of the functions f_1 through f_m so that each of the functions f_i forms a mapping from an n-tuple, at the input, to a single binary value, at the output. The mapping F(x) of the input x can be determined by combining the mapping functions $f_1(x)$ through $f_m(x)$ into one m-tuple.

A special feature is that we can realize the function F by finding a realization for each of the functions f_i. We can now concentrate on the functions f_i which are simpler (considering them one at a time) than the function F. We call the function f_i of x_{n-1} through x_0 (with range {0,1}) a *switching function of n variables*. Later we shall see that we can realize one or more switching functions by means of standard digital ICs, providing switching functions of a limited number of variables. Our design problem can basically be solved in this manner.

Boolean Algebra

and

Switching Algebra

-4.07-

4.1 Boolean algebra and switching algebra

So far we have only encountered truth tables as tools for describing the behaviour of combinational systems. Truth tables become very large for functions of more than a few variables, and thus become quite unmanageable. We feel the need for some other method to describe the behaviour of combinational binary functions. One method is based on a specific type of algebra, the switching algebra. A more common algebra is the Boolean algebra, which was first formulated by George Boole in 1854. Only in 1938 Shannon linked it with switching functions.

SWITCHING ALGEBRA

- A set P with two elements {0,1}
- Two operators + and •
  closed in relation with P
- Postulates = axioms

– P1: the operators are commutative

– P2: the operators are associative

– P3: the operators are distributive

– P4: the operators have a unity element

– P5: each element in the set P
  has an inverse element

–4.08–

COMMUNATIVENESS AND ASSOCIATIVENESS

- P1: the operators are commutative:

$$\forall a,b \in P \quad a+b = b+a$$
$$a \bullet b = b \bullet a$$

- P2: the operators are associative:

$$\forall a,b,c \in P \quad (a+b)+c = a+(b+c)$$
$$(a \bullet b) \bullet c = a \bullet (b \bullet c)$$

- Remarks:

$$(a+b)+c = a+(b+c) = a+(c+b) = (c+a)+b$$
$$(a \bullet b) \bullet c = a \bullet (b \bullet c) = (a \bullet c) \bullet b = -----$$

Here parentheses provide no information; they may be omitted

–4.09–

In this course we shall be confined to switching algebra. Switching algebra (see figure 4.08) is defined by:

- A set of values P holding only two elements which are symbolized by 0 and 1. So all variables belonging to P can only be equal to the symbols 0 and 1, and thus are binary.
- Two operators which we indicate by the + symbol and the · symbol. Those operators are related to the elements of P in such a way that they can be applied to the elements of P, delivering again elements of P.
- Five *postulates* which determine the actions of the operators on the elements. Postulates or axioms are statements which cannot be proven on their own. They are assumed to be true. The postulates and their consequence will be discussed hereafter.

In figure 4.09 we first discuss the postulates P1 and P2 as well as their significance for "daily work". P1: "the operators are commutative", means that the order in which the operands are placed has no significance. So A + B = B + A and A · B = B · A. The arithmetic division for instance, is a non-commutative operator. as we know 4:2 is not equal to 2:4.

P2: "the operations are associative" which means that the order by which an expression, that contains several identical operators, is evaluated has no significance. Calculating A + B first and then taking the result + C gives the same result as A + the result of B + C. The same is true for the · operator. One of the consequences of the postulates P1 and P2 is that parentheses placed in expressions with only one type of operator are unnecessary and can be omitted (figure 4.09 shows some examples).

DISTRIBUTIVENESS

- P3: the operators are distributive

$$\forall a,b,c \in P \quad a\cdot(b+c)=(a\cdot b)+(a\cdot c)$$
$$a+(b\cdot c)=(a+b)\cdot(a+c)$$

- Remarks on priority rule:
  - First • then +
  - Using this rule parentheses can often be removed:

    $a+(b\cdot c)=a+b\cdot c$

    $(a\cdot b)+(a\cdot c)=a\cdot b+a\cdot c$
  - However in the following case this is not possible:

    $(a+b)\cdot(a+c)$
  - Often the • symbol is omitted:

    $a+(b\cdot c)=a+bc \overset{P3}{=} (a+b)(a+c)$

    $a\cdot(b+c)=a(b+c)\overset{P3}{=}ab+ac$

–4.10–

UNITY ELEMENT = NEUTRAL ELEMENT

- P4: for each of the operators • and + there is a unity element in P:
  - An element referred to as 1 for operator •
  - An element referred to as 0 for operator +
  - The following holds for the unity elements:

$$\forall a \in P \quad a+0=a$$
$$a\cdot 1=a$$

- Remark:
  - The unity elements 0 and 1 are unique
  - Proof:

    Assume $0_1,0_2 \in P$ are <u>two</u> elements related to the operator +

    Then $\forall a \in P$ holds $a+0_1=a$

    Take $a=0_2$, then $0_2+0_1=0_2$

    However also $\forall a \in P$ holds $a+0_2=a$

    Take $a=0_1$, then $0_1+0_2=0_1$

    Accordingly

    $$0_2=0_2+0_1=0_1+0_2=0_1$$

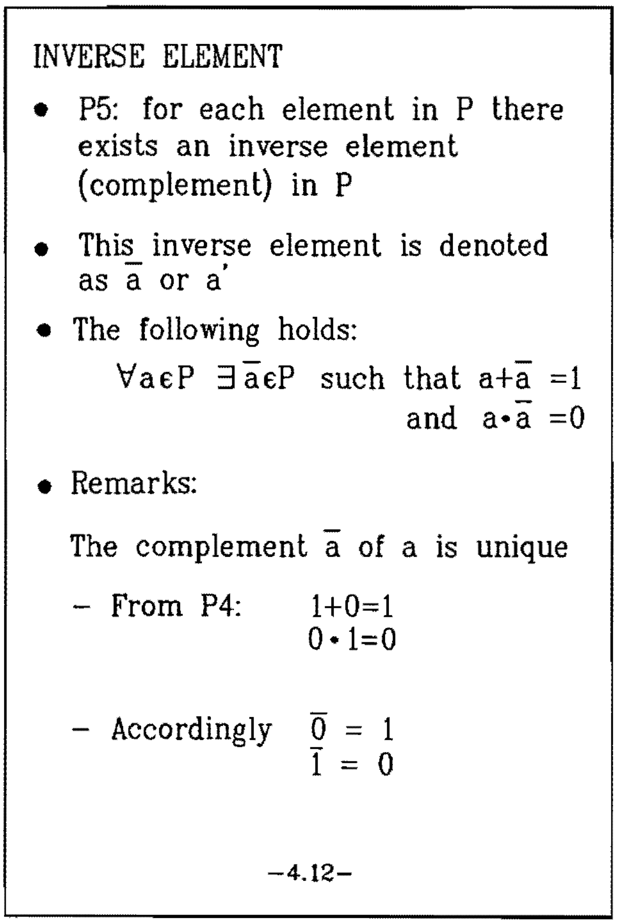
    $$\boxed{0_2=0_1=0}$$

–4.11–

P3: "the operators are distributive" (figure 4.10) means that the dot-operator can be distributed over the left and right operands of the plus-operator, and that the plus-operator can be distributed over the left and right operands of the dot-operator. Figure 4.10 shows some examples.

Now we shall attach priorities to both operators. We shall give the dot-operator a higher priority than the plus-operator. This means that we can frequently omit the parentheses when writing expressions. That is to say we can omit the parentheses that enclose a dot-operator. Indeed, the dot-operator is evaluated first because of the given priority rule. Figure 4.10 shows a number of examples. Frequently we go one step further and omit the dot-operator symbol. In this case we usually write variables juxtaposed in sub-expressions. Figure 4.10 shows some examples.

P4: For both of the dot and plus operators there is at least one unity element in P, denoted by 1 and 0 respectively (see figure 4.11). We also say that 0 is the neutral element of the plus-operator, and that 1 is the neutral element of the dot-operator. This means that an operation with a second argument equal to the unit element delivers the first argument as a result. Thus a + 0 = a and a · 1 = a. We see from figure 4.11 that the unit elements 0 and 1 are unique. That is to say: there are no two unit elements in P with the same attribute. The proof that 0 is a unit element with respect to the plus-operator is shown in figure 4.11. The proof for the element 1 goes along similar lines.



INVERSE ELEMENT

- P5: for each element in P there exists an inverse element (complement) in P
- This inverse element is denoted as $\bar{a}$ or a'
- The following holds:

  $\forall a \in P\ \exists \bar{a} \in P$ such that $a+\bar{a}=1$ and $a\cdot\bar{a}=0$

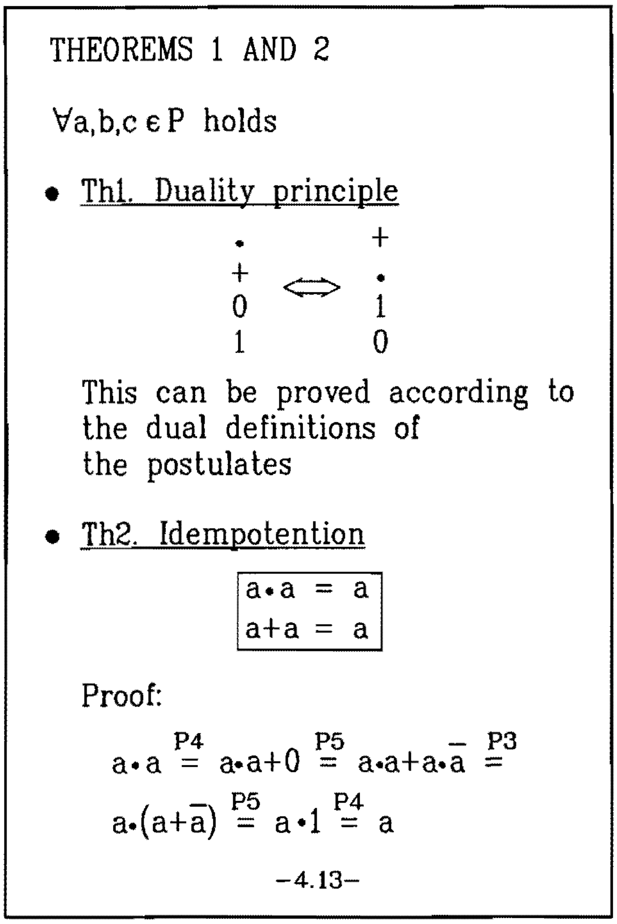
- Remarks:

  The complement $\bar{a}$ of a is unique

  – From P4: $1+0=1$
 $0\cdot 1=0$

  – Accordingly $\bar{0}=1$
 $\bar{1}=0$

–4.12–



THEOREMS 1 AND 2

$\forall a,b,c \in P$ holds

- Th1. Duality principle

$$\begin{array}{ccc} \cdot & & + \\ + & \Leftrightarrow & \cdot \\ 0 & & 1 \\ 1 & & 0 \end{array}$$

This can be proved according to the dual definitions of the postulates

- Th2. Idempotention

$$\boxed{\begin{array}{l} a\cdot a = a \\ a+a = a \end{array}}$$

Proof:

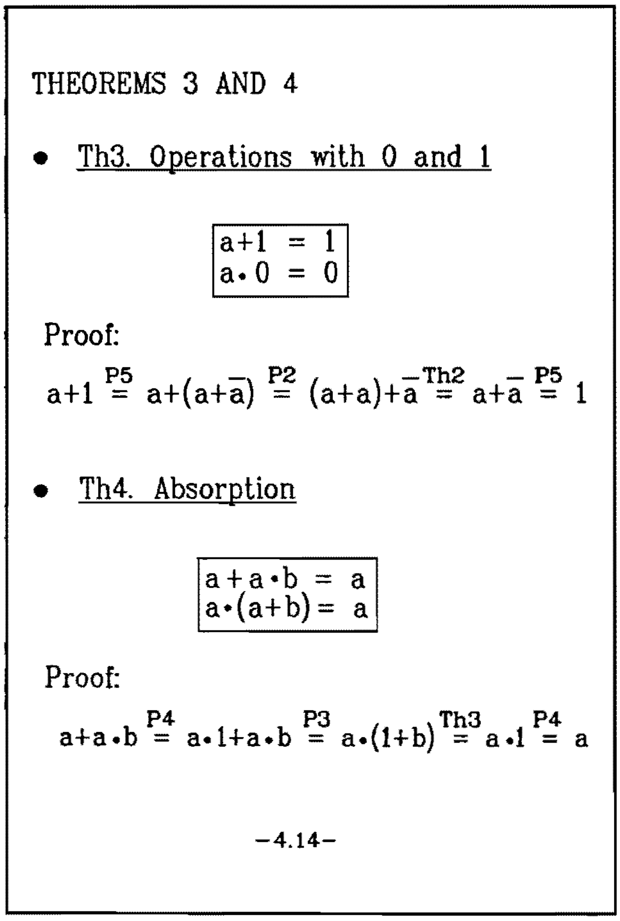
$$a\cdot a \overset{P4}{=} a\cdot a+0 \overset{P5}{=} a\cdot a+a\cdot\bar{a} \overset{P3}{=} a\cdot(a+\bar{a}) \overset{P5}{=} a\cdot 1 \overset{P4}{=} a$$

–4.13–

P5 states: for each element in P there exists a complement in P, also called inverse. If we express this inverse with $\bar{a}$ or a', then it is valid (for the inverse of a) that a + a' = 1 and a · a' = 0. See also figure 4.12. We also note that complement of a (i.e. a') is unique. Therefore, each element in P has exactly one complement. We shall not discuss the proof of this assertion. Another important aspect of the postulate P5 is that the inverse of element 0 is equal to the element 1, and that the inverse of 1 is 0. The prof of this property is shown in figure 4.12.

From the postulates, that can not be proven on their own, we can derive a number of consequences. We have already mentioned some of these results; other results deserve some more attention. Therefore, they are expressed as *theorems*. In the following figures we shall formulate 7 theorems. We shall discuss these theorems briefly without considering the derivation of their proof. These figures show for each theorem at least one prof, the proof of the rest of the theorem can be simply derived from the given part. This will be left to a number of exercises.

Figure 4.13 shows, as the first theorem, the duality principle. We can formulate this theorem as follows: Each algebraic switching expression remains valid when we exchange the plus and dot operators and at the same time the unit elements 0 and 1. The proof of the duality principle follows immediately from the dual definition of the postulates. An expression that is formulated by the exchange of operators and the unit element of another expression is called a *dual expression.* When considering the postulates from P1 to P5 we always find expressions that are dual to each other.
Figure 4.13 shows theorem 2, the *idempotent* property. "Idempotent" literally means the same power or the same ability. Under the idempotent property we understand that the expression a · a is the same as a, i.e. can be replaced by a. The same is true for a + a.



THEOREMS 3 AND 4

- Th3. Operations with 0 and 1

$$\boxed{\begin{aligned} a+1 &= 1 \\ a\bullet 0 &= 0 \end{aligned}}$$

Proof:

$$a+1 \overset{P5}{=} a+(a+\bar{a}) \overset{P2}{=} (a+a)+\bar{a} \overset{Th2}{=} a+\bar{a} \overset{P5}{=} 1$$

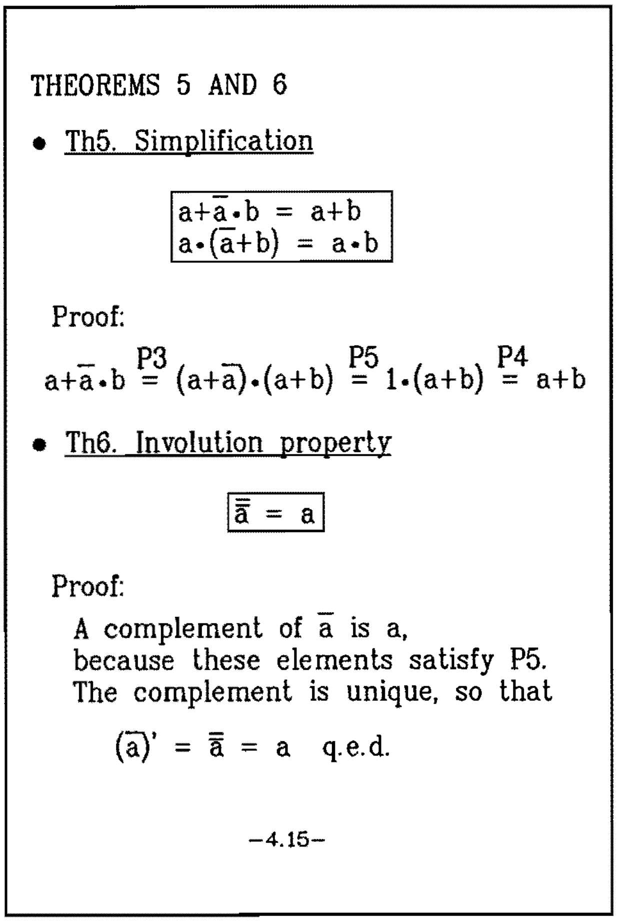
- Th4. Absorption

$$\boxed{\begin{aligned} a+a\bullet b &= a \\ a\bullet(a+b) &= a \end{aligned}}$$

Proof:

$$a+a\bullet b \overset{P4}{=} a\bullet 1+a\bullet b \overset{P3}{=} a\bullet(1+b) \overset{Th3}{=} a\bullet 1 \overset{P4}{=} a$$

–4.14–



THEOREMS 5 AND 6

- Th5. Simplification

$$\boxed{\begin{aligned} a+\bar{a}\bullet b &= a+b \\ a\bullet(\bar{a}+b) &= a\bullet b \end{aligned}}$$

Proof:

$$a+\bar{a}\bullet b \overset{P3}{=} (a+\bar{a})\bullet(a+b) \overset{P5}{=} 1\bullet(a+b) \overset{P4}{=} a+b$$

- Th6. Involution property

$$\boxed{\bar{\bar{a}} = a}$$

Proof:

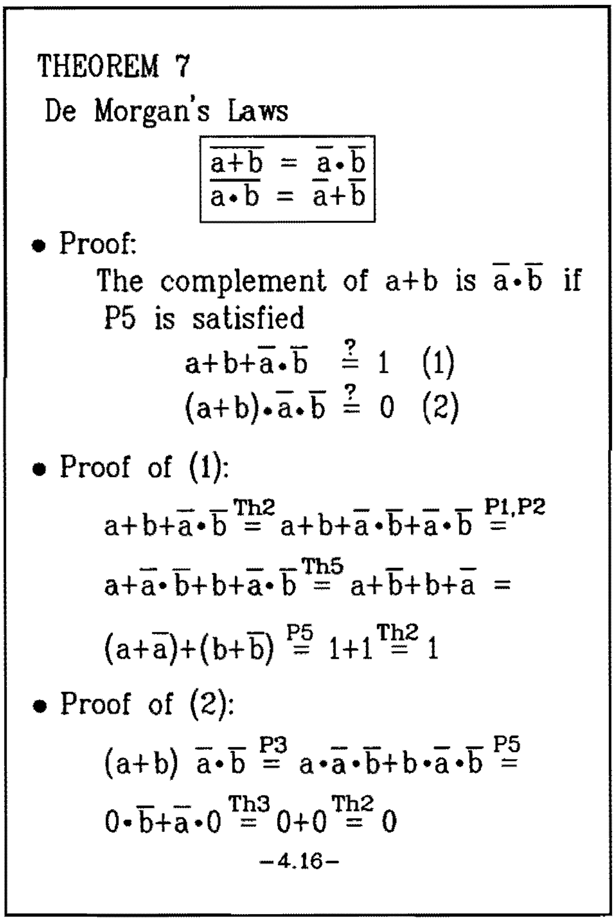
A complement of $\bar{a}$ is a,
because these elements satisfy P5.
The complement is unique, so that

$(\bar{a})' = \bar{\bar{a}} = a$ q.e.d.

–4.15–

Theorem 3 in figure 4.14 shows some more properties of operations with the elements 0 and 1. Compare these with the properties of the unit elements as formulated in postulate P4. Postulate P4 and theorem 3 lead to four different operations on the elements 1 and 0. Theorem 4 is known as the "absorption" theorem. We see that the value of the variable b has no effect on the value of the whole expression, and hence it can be omitted. The absorption theorem can be used to simplify expressions, i.e. to eliminate variables from the expression.

Another form of simplification of expressions is shown in theorem 5 (see figure 4.15). In this theorem we see that the inverse of a is redundant in some expressions, and thus can be omitted. Theorem 6 is known as the "involution property". This theorem shows that the inverse of the inverse of an element delivers the original value of the element. Thus complementing a variable twice produces the original variable again.



THEOREM 7

De Morgan's Laws

$$\boxed{\begin{array}{l}\overline{a+b} = \overline{a}\bullet\overline{b} \\ \overline{a\bullet b} = \overline{a}+\overline{b}\end{array}}$$

- Proof:

  The complement of a+b is $\overline{a}\bullet\overline{b}$ if P5 is satisfied

  $$a+b+\overline{a}\bullet\overline{b} \overset{?}{=} 1 \quad (1)$$

  $$(a+b)\bullet\overline{a}\bullet\overline{b} \overset{?}{=} 0 \quad (2)$$

- Proof of (1):

  $$a+b+\overline{a}\bullet\overline{b} \overset{Th2}{=} a+b+\overline{a}\bullet\overline{b}+\overline{a}\bullet\overline{b} \overset{P1,P2}{=}$$

  $$a+\overline{a}\bullet\overline{b}+b+\overline{a}\bullet\overline{b} \overset{Th5}{=} a+\overline{b}+b+\overline{a} =$$

  $$(a+\overline{a})+(b+\overline{b}) \overset{P5}{=} 1+1 \overset{Th2}{=} 1$$

- Proof of (2):

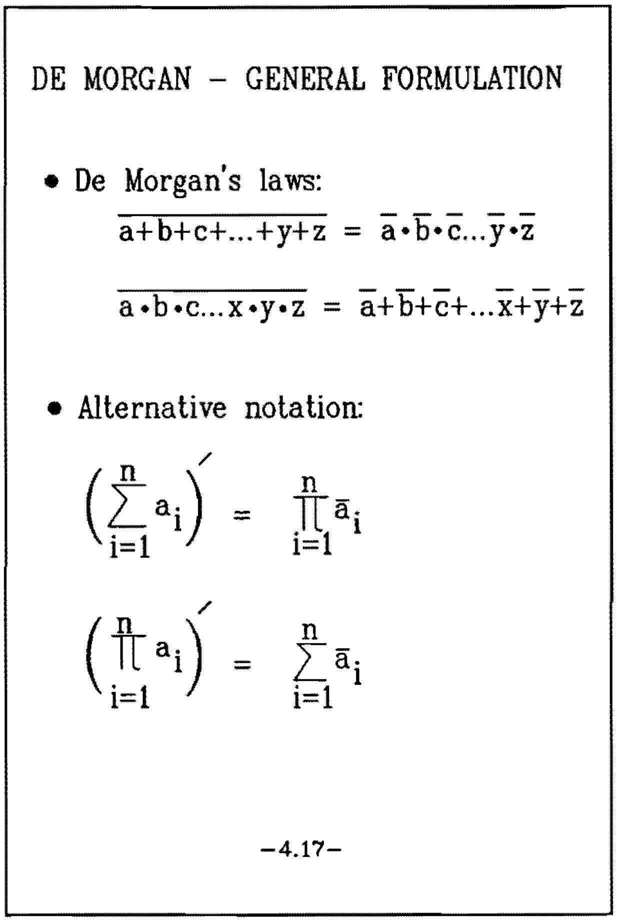
  $$(a+b)\ \overline{a}\bullet\overline{b} \overset{P3}{=} a\bullet\overline{a}\bullet\overline{b}+b\bullet\overline{a}\bullet\overline{b} \overset{P5}{=}$$

  $$0\bullet\overline{b}+\overline{a}\bullet 0 \overset{Th3}{=} 0+0 \overset{Th2}{=} 0$$

-4.16-

In figure 4.16 we finally see theorem 7, the De Morgan laws. These laws play a very important role in the applications of switching algebra. They show that the inverse of an expression with the plus-operator is equal to an expression with the dot-operator and inverted variables. The reverse is valid for expressions with the dot-operator. Figure 4.16 shows De Morgan's laws formulated for two variables. Figure 4.17 gives a more general formulation of De Morgan's laws. The inverse of an expression with only plus-operators gives an expression with only dot-operators and inverted variables. The same holds for the inverse of expressions with only a dot-operator.

The symbols we use for the plus and dot operators also have a meaning in normal algebra. We have, in the meanwhile, seen so much of switching algebra that we realize that the related operators are completely different. Therefore, we have until now used the terms plus and dot-operators consistently. This is not really practical. In daily practice we also carelessly use words such as *sum* and *product* to denote the dot and plus operators respectively. We shall, from now on, follow this convention.



DE MORGAN – GENERAL FORMULATION

- De Morgan's laws:

$$\overline{a+b+c+...+y+z} = \bar{a}\bullet\bar{b}\bullet\bar{c}...\bar{y}\bullet\bar{z}$$

$$\overline{a\bullet b\bullet c...x\bullet y\bullet z} = \bar{a}+\bar{b}+\bar{c}+...\bar{x}+\bar{y}+\bar{z}$$

- Alternative notation:

$$\left(\sum_{i=1}^{n} a_i\right)' = \prod_{i=1}^{n} \bar{a}_i$$

$$\left(\prod_{i=1}^{n} a_i\right)' = \sum_{i=1}^{n} \bar{a}_i$$

–4.17–

The De Morgan laws could also be formulated as follows: the negation of the complement of a sum of a number of variables is equal to the product of the negation of the variables. Now we can understand the notation of figure 4.17 where we have used the sum or Σ sign to denote the sum of a number of variables, and the product or π sign to denote the product of a number of variables. By using this notation we can write De Morgan's laws in a short-form.

Problem 4.1 (4.2)

Show by means of switching algebra postulates and theorems the validity of the following:

a. $\bar{a}b+a(\bar{b}+c)+a\bar{b}c = \bar{a}b+a(\bar{b}+c)$

b. $ab\bar{c}+\bar{a}bc+\bar{a}b\bar{c}+abc+\bar{a}\bar{b}c+a\bar{b}c = b+c$

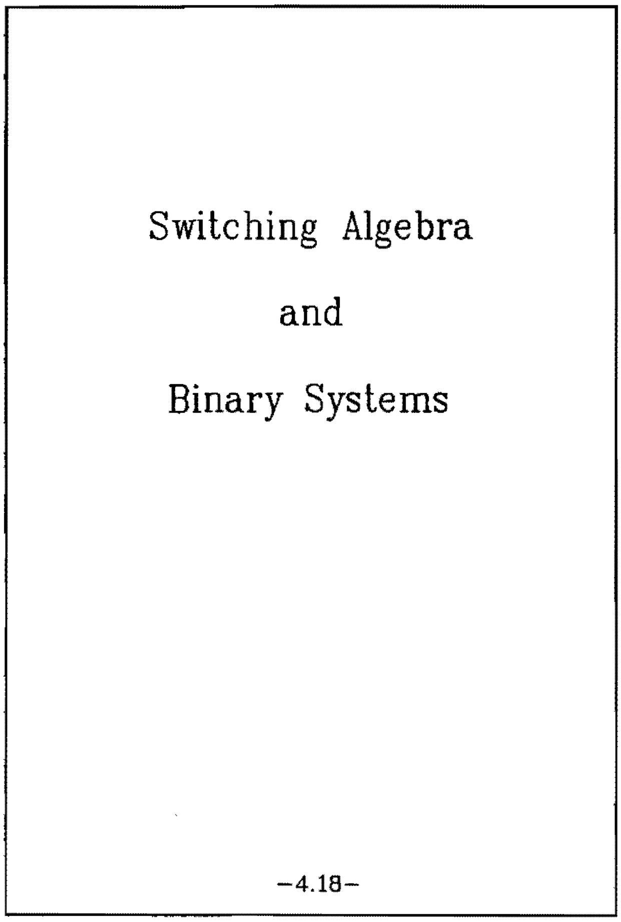
c. $\overline{\overline{\overline{xy}z} + \bar{x}y\bar{z}} = (\bar{x}+\bar{y})z$

d. $x+\bar{x}y+z(x+y) = x+y$

e. $(a+\bar{c})(a+b)(b+c) = (a+\bar{c})(b+c)$

f. $xy+\bar{x}z+yz = xy+\bar{x}z$
Compare with problem e; what do you remark?

g. $wx+xy+\bar{x}\bar{z}+w\bar{y}z = xy+\bar{x}\bar{z}+w\bar{y}$
(Application of problem f.)



Switching Algebra

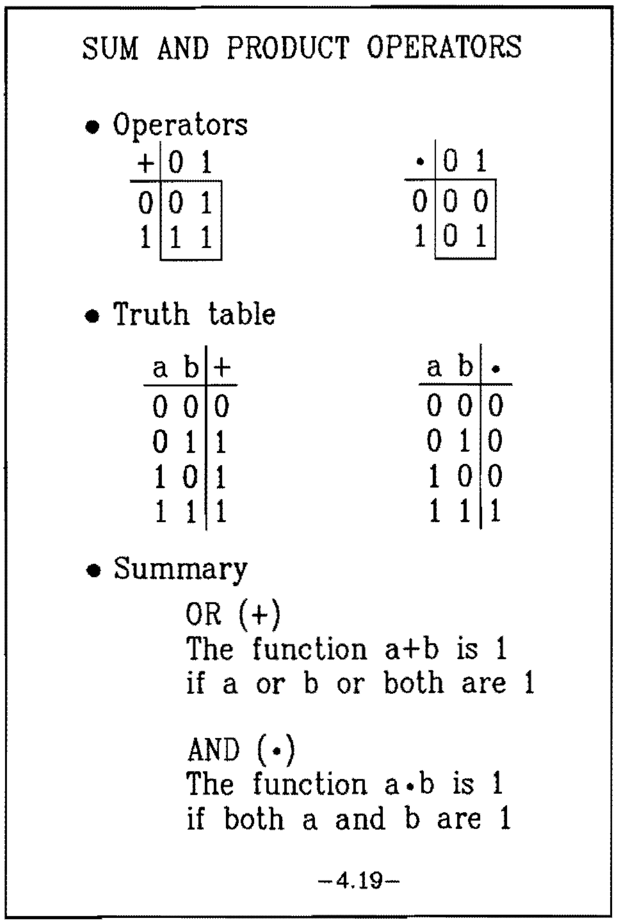
and

Binary Systems

−4.18−

4.2 Switching algebra and binary systems

As a motivation for the introduction and use of switching algebra we have mentioned that we need, next to truth tables, a more formal and compact specification for the behaviour of binary systems. We have discussed the principles of switching algebra. The following step is to make a connection between switching algebra and the behaviour of binary systems, such as the one given by, for example, a truth table.



SUM AND PRODUCT OPERATORS

- Operators

$$\begin{array}{c|cc} + & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 1 \end{array} \qquad \begin{array}{c|cc} \cdot & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}$$

- Truth table

a	b	+
0	0	0
0	1	1
1	0	1
1	1	1

a	b	•
0	0	0
0	1	0
1	0	0
1	1	1

- Summary

OR (+)
The function a+b is 1
if a or b or both are 1

AND (•)
The function a•b is 1
if both a and b are 1

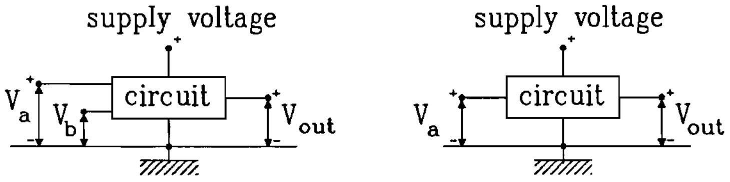
−4.19−

As a first step we shall consider figure 4.19; there we examine the result of applying the sum and product operators on the elements 0 and 1. For each of the two operators we can, using postulate 4 and theorem 3, construct a matrix. From this matrix we see, for example, that the sum of the elements 0 and 1 is always 1. We also see that the product of two elements is only equal to 1 if both element are equal to 1. From such a matrix of the sum or product operators we can simply deduce the truth table for such an operation. Notice that the truth table on one hand introduces the effect of the sum or product operator, and on the other hand is a behavioural description of a binary system that executes this sum or product operation. The behaviour of a binary system is given by a functional relation, i.e. a function. In the case of the sum operator this function is called the *or-function.* The or-function is defined by:

$a + b = 1$ if a or b or both are equal to 1.

Finally the product operator, i.e. the and function is, defined by: $a \cdot b = 1$ if both a and b are equal to 1.

ELECTRONIC CIRCUITS



circuit 1

V_a	V_b	V_{out}
L	L	L
L	H	H
H	L	H
H	H	H

circuit 2

V_a	V_b	V_{out}
L	L	L
L	H	L
H	L	L
H	H	H

circuit 3

V_a	V_{out}
L	H
H	L

For TTL L < 0.8 Volts and H > 2.4 Volts

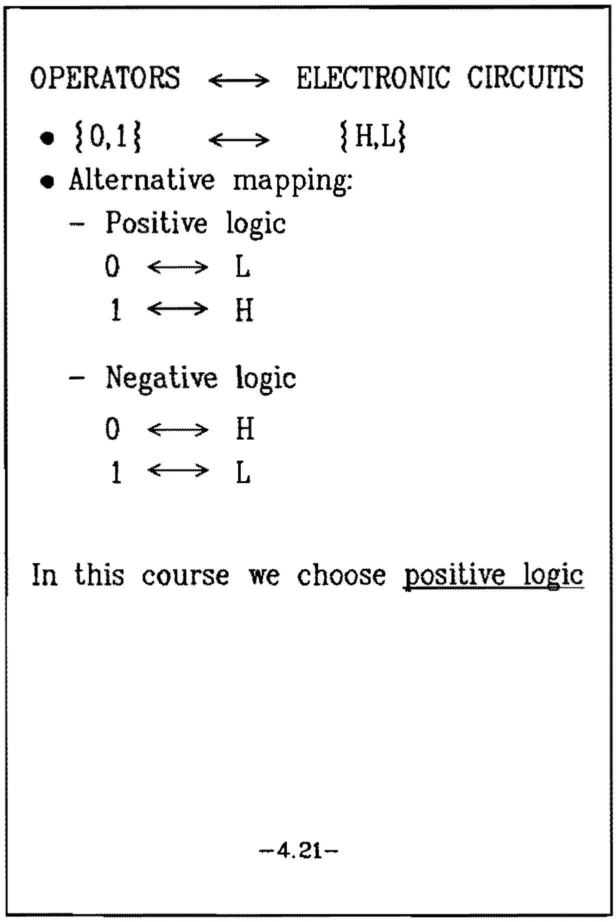
-4.20-

In figure 4.20 we show symbolically that we can make a circuit, using electronic devices (such as transistors and resistors) that are integrated on a silicon chip, that shows the following behaviour. For circuit 1 it appears that when both the input voltages V_a and V_b have a low value the output voltage V_{out} also has a low value. The output voltage V_{out} will have a high value if one or both of the input voltage have a high value. The definition of what is a low or high voltage depends on a number of factors such as the height of the feeding voltage, the used building blocks etc. In the frequently used family of building blocks, the TTL family or the Transistor Transistor Logic family, L stands for a voltage < 0.8 Volt and H for a voltage > 2.4 Volt. In some modern CMOS building blocks these numbers are different (CMOS = Complementary Metal Oxide Semiconductor).

Another frequently used circuit exhibits the behaviour shown in figure 4.20 as circuit 2. The output voltage V_{out} remains low if at least one of the inputs V_a or V_b has a low level. The output value is only high if both input values are high.

Finally we have a third important circuit with only one input and one output. The output voltage is high if the input voltage is low, and the output voltage is low if the input voltage is high.

Of course voltages in an electronic circuit are analog; i.e. they may take many possible values. Actually, by defining voltage thresholds we can consider voltages lower than a threshold value and voltage higher than another threshold value. Thus, we can consider "low" and "high" voltages. By designing our circuit such that the output voltages are also lower or higher than certain thresholds we create circuits where the input and output are dual-valued (i.e. binary). These inputs and outputs only have high or low voltage values. We shall not concern ourselves with knowing the precise value of such voltages.



OPERATORS ⟷ ELECTRONIC CIRCUITS

- {0,1} ⟷ {H,L}
- Alternative mapping:
  - Positive logic

    0 ⟷ L

    1 ⟷ H
  - Negative logic

    0 ⟷ H

    1 ⟷ L

In this course we choose positive logic

–4.21–

Figure 4.21 shows on one hand switching algebra with operators that act on the elements 0 and 1, and on the other hand that electronic circuit can be realized where the output voltages are high or low. We shall eventually make use of these electronic circuit as realizations of the related operator functions. Therefore it is necessary that we map the element {0,1} to the elements {H,L}. Such a mapping can be done in two ways. First we can map 0 to L and 1 to H. We are then considering *positive logic.* We can really, without any problem, map 0 on the high voltage level and 1 on the low voltage level. We are then considering *negative logic.* In principle, there is no advantage for one or the other mapping. However, we must make a choice to avoid confusion. In this course we shall always use positive logic; thus we shall only make use of the mapping: 0 becomes a low voltage, and 1 becomes a high voltage.

"AND" AND "OR" GATES

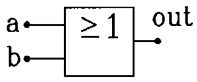
circuit 1

V_a	V_b	V_{out}
L	L	L
L	H	H
H	L	H
H	H	H

⇒

a	b	out
0	0	0
0	1	1
1	0	1
1	1	1

(or-gate)



out = a+b

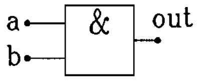
circuit 2

V_a	V_b	V_{out}
L	L	L
L	H	L
H	L	L
H	H	H

⇒

a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

(and-gate)



out = a•b

-4.22-

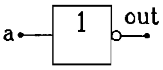
INVERTER

V_a	V_{out}
L	H
H	L

⇒

a	out
0	1
1	0

inverter

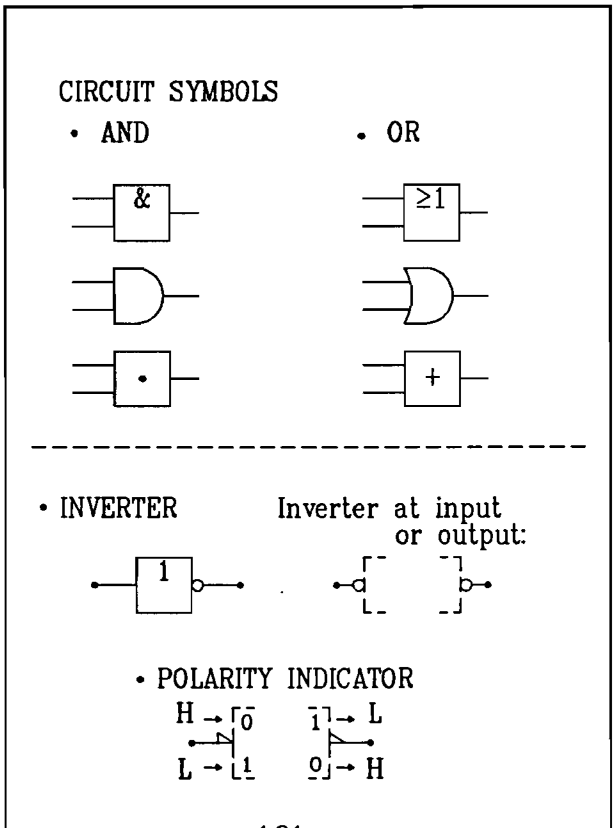


out = $\bar{a}$

-4.23-

Figure 4.22 shows what this implies for circuit 1 and circuit 2. Replacing L by 0 and H by 1 leads to the truth table shown in figure 4.22. By comparing this with the truth table of the and-function and or-function, shown in figure 4.19 we see that they are identical. Circuit 1 is thus a realization of the or-function. This circuit is also called an *or-gate.* In schematics we use the shown symbol (figure 4.22) for or-gates. The truth table related to circuit 2 is equivalent to the truth table of the and-function. Circuit 2 is thus called an *and-gate.* Figure 4.22 shows the and-gate symbol.

For circuit 3 we find the truth table shown in figure 4.23. We see that the output "OUT" is equal to the inverse of a. Therefore, we call circuit 3 an *inverter.* Figure 4.23 also shows the inverter symbol. This symbol is, in fact, composed of two parts. The rectangular block with a 1 in the middle acts as an amplifier (buffer). The inverting function is denoted by the small circle that is shown at the output close to the rectangle.



-4.24-

In this course we shall make use of the IEC (International Electrotechnical Commission) standard schematic symbols. They are described in the standard document IEC 617-12. A summary can also be found in Appendix C of this survey. Figure 4.24 again shows the symbols we use for and-gates and or-gates, among other frequently used gate symbols. We shall see these symbols in all sorts of schematics. We shall make two remarks about the use of these symbols. We have seen that the inversion function (an inverter) is expressed by a circle. To specify an inverted behaviour in a schematic we do not have to draw the complete inverter; it is sufficient to draw the circle symbol at an input or output of a gate. We shall later give some examples.

The second remark concerns the use of negative logic mixed with positive logic. In the standard symbol set of the schematic symbols of digital symbols, we find the so-called *polarity indicators*. A polarity indicator is shown as a small triangle at the input or output (see figure 4.24). A polarity indicator shows that a high input voltage is translated into an internal logic 0 and a low input voltage is translated into an internal logic 1. We act as if the circuit performs its operations to these logical values. The polarity indicator on the output shows that an internal logical 1 is translated to a low voltage at the output port of the circuit. A logical 0 is also translated to a high voltage at the output of the circuit.

SWITCHING FUNCTION AND TRUTH TABLE

$f(a,b) = a\bullet\overline{b}+\overline{a}\bullet b$

$f(0,0) = 0\bullet\overline{0}+\overline{0}\bullet 0 = 0\bullet 1+1\bullet 0 = 0+0 = 0$

$f(0,1) = 0\bullet\overline{1}+\overline{0}\bullet 1 = 0\bullet 0+1\bullet 1 = 0+1 = 1$

$f(1,0) = 1\bullet\overline{0}+\overline{1}\bullet 0 = 1\bullet 1+0\bullet 0 = 1+0 = 1$

$f(1,1) = 1\bullet\overline{1}+\overline{1}\bullet 1 = 1\bullet 0+0\bullet 1 = 0$

a	b	f
0	0	0
0	1	1
1	0	1
1	1	0

f is 1 if (a is 0 and b is 1) or (a is 1 and b is 0)

f is 1 if ($\overline{a}$ and b are both 1) or (a and $\overline{b}$ are both 1)

f is 1 if ($\overline{a}\bullet b$ is 1) or ($a\bullet\overline{b}$ is 1)

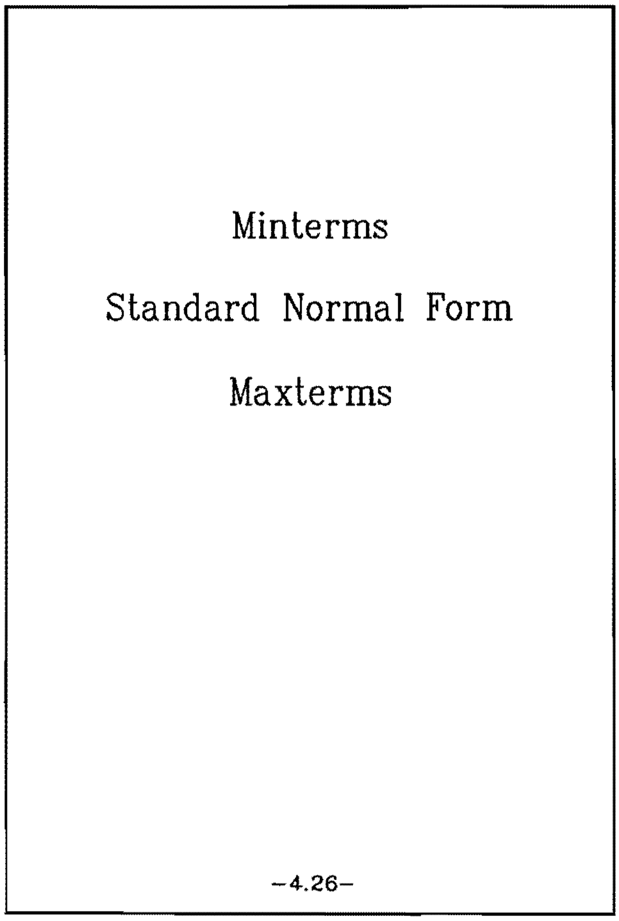
f is 1 if $\overline{a}\bullet b + a\bullet\overline{b}$ is 1

$$f = \overline{a}\bullet b + a\bullet\overline{b}$$

–4.25–

We have used switching algebra (with the help of sum and product operators) as a method of describing switching functions. We have seen that we used certain methods to realize these operations (in the and-gate and the or-gate). Later we shall see that each arbitrary switching algebraic expression can be realized by the right combination of and-gates and or-gates. But can we realize any arbitrary combinational binary function? In other words what is the relation between the truth table of a combinational binary function and switching algebra? Figure 4.25 further examines this relation.

Starting from an algebraic switching expression or a switching function we can derive the truth table by substituting the function's n variables by all possible n-tuples; this is done by filling in all the possible variables' values. This is shown in figure 4.25 for a function of two variables a and b. With the help of our postulates and theorems we can reduce these constant expressions to a single 0 or 1 value. This is then the value of the specific function for this binary n-tuple. In this way we can find the switching function of the truth table. The other way around, we can make an assertion about the value of the function from the truth table. This assertion we can then rewrite to a switching function of the variables. From the truth table of figure 4.24 we derive that the function f = 1 if a = 0 and b = 1 or if a = 1 and b = 0. This statement can be re-derived, as shown, to a switching algebraic expression. We shall return in the following paragraphs to the relation between the truth table and switching functions.

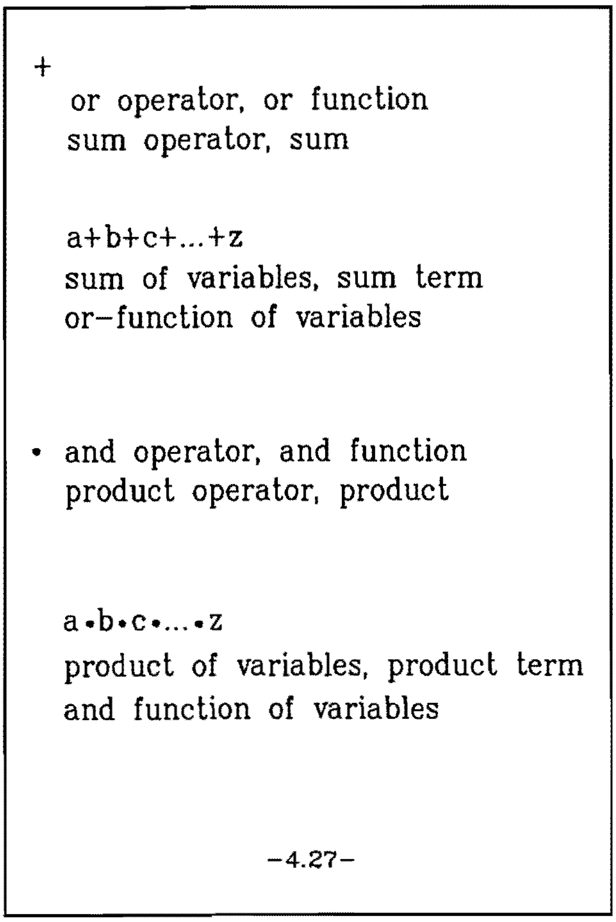


Minterms

Standard Normal Form

Maxterms

-4.26-



+
or operator, or function
sum operator, sum

a+b+c+...+z
sum of variables, sum term
or–function of variables

• and operator, and function
product operator, product

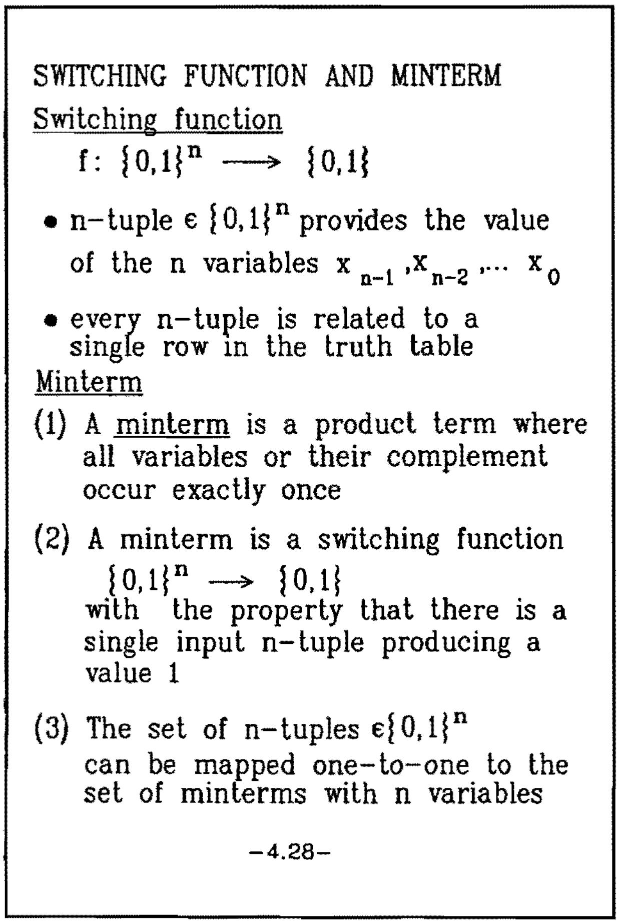
a•b•c•...•z
product of variables, product term
and function of variables

-4.27-

4.3 Minterms, standard normal form and maxterms

In this section we shall formally determine that the behaviour of all combinational binary functions can be defined by, at least, an algebraic expression. In a following chapter we shall see that this also means that we can in principle realize all combinational binary functions.
The clue is called the *standard normal* form.

First we shall present (see figure 4.27) a number of frequently used expressions. Instead of using the terms plus-operator or sum-operator we frequently use the terms or-operator, or-function, sum etc. We frequently describe an expression such as a + b + c + z by: sum of variables, sum term or or-function of variables. We also refer to the dot-operator or the product operator as the and-operator or simply the product. We shall also use expressions such as: the product of variables, a product term, and-function of variables etc.



SWITCHING FUNCTION AND MINTERM

Switching function

$f: \{0,1\}^n \longrightarrow \{0,1\}$

- n-tuple $\in \{0,1\}^n$ provides the value of the n variables $x_{n-1}, x_{n-2}, \ldots x_0$
- every n-tuple is related to a single row in the truth table

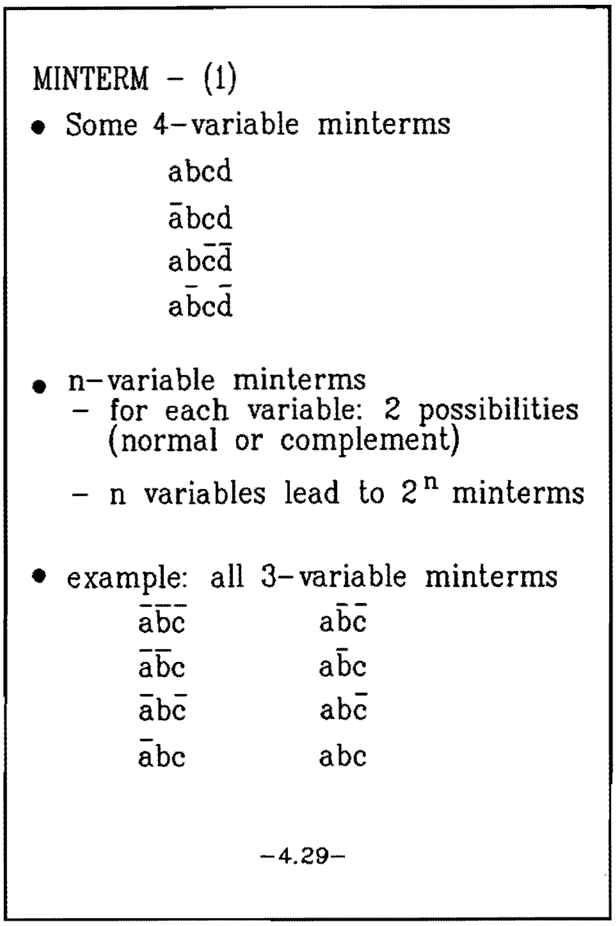
Minterm

(1) A minterm is a product term where all variables or their complement occur exactly once

(2) A minterm is a switching function $\{0,1\}^n \longrightarrow \{0,1\}$ with the property that there is a single input n-tuple producing a value 1

(3) The set of n-tuples $\in \{0,1\}^n$ can be mapped one-to-one to the set of minterms with n variables

-4.28-



MINTERM – (1)

- Some 4-variable minterms

  $abcd$

  $\bar{a}bcd$

  $ab\bar{c}\bar{d}$

  $a\bar{b}c\bar{d}$

- n-variable minterms
  - for each variable: 2 possibilities (normal or complement)
  - n variables lead to 2^n minterms

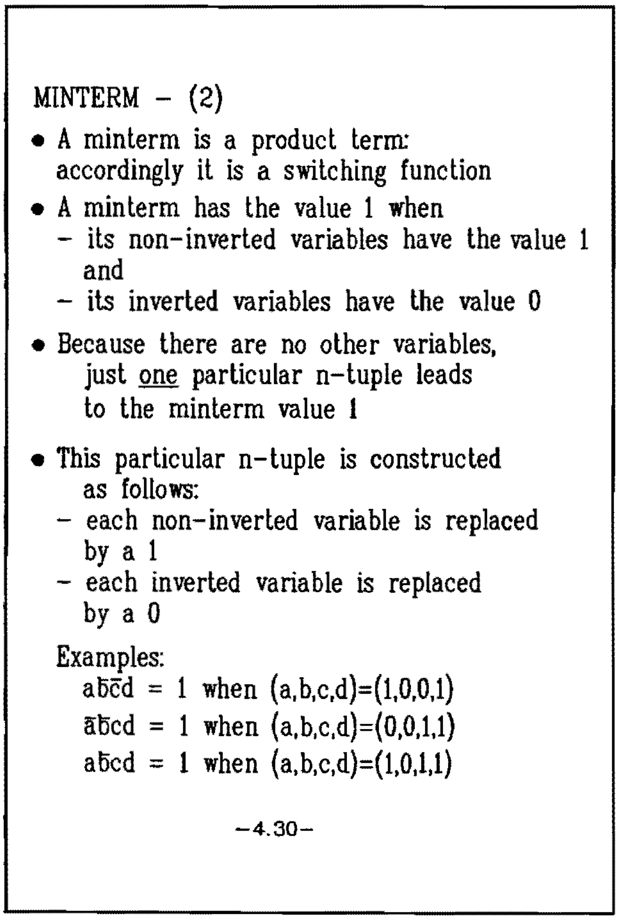
- example: all 3-variable minterms

$\bar{a}\bar{b}\bar{c}$	$a\bar{b}\bar{c}$
$\bar{a}\bar{b}c$	$a\bar{b}c$
$\bar{a}b\bar{c}$	$ab\bar{c}$
$\bar{a}bc$	abc

-4.29-

We have defined a switching function as a mapping from a binary n-tuple to a value of the set {0,1}. On the other hand we have defined a switching function as function of n variables x_{n-1} to x_0. The choice of an input n-tuple (from the set of possible input n-tuples) is connected to assigning a 0 or 1 value to each of the variables x_{n-1} to x_0 (see figure 4.28). We have also seen that each n-tuple corresponds to exactly one row in the truth table. Keeping all of these three things in mind we arrive at the three assertions in figure 4.28.

1. We define a *minterm* as a product term where all variables or their complements occur exactly once. Figure 4.29 shows a number of minterm examples. Notice that we may omit the product operator symbol. Because each variable or its complement occurs exactly once we can compose 2^n minterms from n variables. Thus, for 3 variables we have 8 terms. Figure 4.29 shows all of these 8 minterms.



MINTERM – (2)

- A minterm is a product term: accordingly it is a switching function
- A minterm has the value 1 when
  - its non-inverted variables have the value 1 and
  - its inverted variables have the value 0
- Because there are no other variables, just <u>one</u> particular n-tuple leads to the minterm value 1
- This particular n-tuple is constructed as follows:
  - each non-inverted variable is replaced by a 1
  - each inverted variable is replaced by a 0

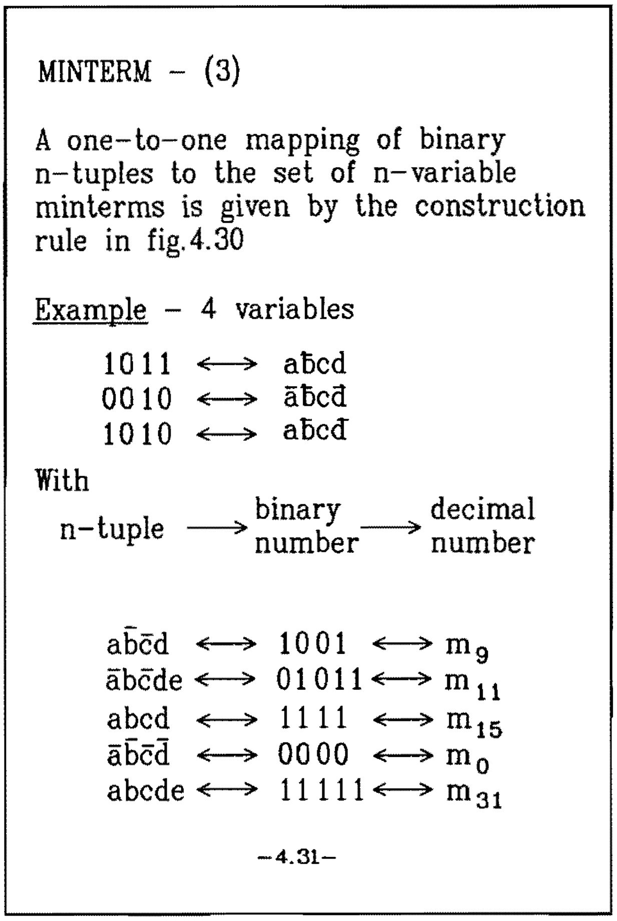
Examples:

$a\bar{b}\bar{c}d$ = 1 when (a,b,c,d)=(1,0,0,1)

$\bar{a}\bar{b}cd$ = 1 when (a,b,c,d)=(0,0,1,1)

$a\bar{b}cd$ = 1 when (a,b,c,d)=(1,0,1,1)

–4.30–



MINTERM – (3)

A one-to-one mapping of binary n-tuples to the set of n-variable minterms is given by the construction rule in fig.4.30

<u>Example</u> – 4 variables

$1011 \longleftrightarrow a\bar{b}cd$

$0010 \longleftrightarrow \bar{a}\bar{b}c\bar{d}$

$1010 \longleftrightarrow a\bar{b}c\bar{d}$

With

$\text{n-tuple} \longrightarrow \begin{array}{c}\text{binary}\\\text{number}\end{array} \longrightarrow \begin{array}{c}\text{decimal}\\\text{number}\end{array}$

$a\bar{b}\bar{c}d \longleftrightarrow 1001 \longleftrightarrow m_9$

$\bar{a}b\bar{c}de \longleftrightarrow 01011 \longleftrightarrow m_{11}$

$abcd \longleftrightarrow 1111 \longleftrightarrow m_{15}$

$\bar{a}\bar{b}\bar{c}\bar{d} \longleftrightarrow 0000 \longleftrightarrow m_0$

$abcde \longleftrightarrow 11111 \longleftrightarrow m_{31}$

–4.31–

2. A minterm is a switching function of a binary n-tuple to a binary variable, with the extra feature that *exactly one n-tuple yields the function value 1.* A minterm is a product term and thus it is a switching function (see figure 4.30). A minterm (and thus a function) has the value 1 if all variables that appear in a non-inverted form in the minterm have the value 1, and all inverted variables have the value 0. This is the only case for which all the operands of the product operators are equal to 1 yielding a result equal to 1. By doing so we have given all the variables a value corresponding to exactly one n-tuple. Figure 4.30 shows how we can construct this special n-tuple for a minterm.

3. The set of all binary n-tuples can be mapped one-to-one to the set of all minterms with n variables. The indicated mapping is given by the so-called construction rule of the n-tuple from the corresponding minterm. Figure 4.31 shows a number of examples.

A minterm is defined by its algebraic expression. Actually, the relation between a minterm and a binary n-tuple leads to another frequently used notational from. A binary n-tuple (which is related to a minterm) can be seen as a binary number with a value. The related decimal number is frequently used to express the minterm. Figure 4.31 shows a number of examples. With minterm m_9 we mean the minterm associated with the quadruple 1001, that is the minterm ab'c'd. Notice now that the place of the variable in the n-tuple plays an important role. This is due to the fact that the positions of the ones and zeros in the binary n-tuple are coupled to weights. In all our examples the variable a will be related to the bit with the highest weight in the n-tuple, i.e. the most significant one.

EXAMPLE

- Function:

$$f: \{0,1\}^3 \longrightarrow \{0,1\}$$

- Truth table:

a b c	f
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	1
1 0 0	1
1 0 1	0
1 1 0	0
1 1 1	1

└ function value

└ n-tuples (triples)

- f is 1 for triples 001, 011, 100, 111
- f is 1 if minterm $\bar{a}\bar{b}c$ or $\bar{a}bc$ or $a\bar{b}\bar{c}$ or abc is 1
- f is 1 if $\bar{a}\bar{b}c+\bar{a}bc+a\bar{b}\bar{c}+abc$ is 1

-4.32-

What did we achieve with these three points? Figure 4.32 shows a truth table of a function with three variables. From this truth table, we see that the function value 1 results from the input triples 011, 100 and 111. Actually, with the help of the previously formulated relation between n-tuples (in this case triples) and minterms we can say that the function value is 1 when one of the minterms a'b'c, a'bc, ab'c' or abc is equal to 1. We know that at most one of the mentioned minterms can be 1, and that this is exclusively so when the variables a, b and c have the values shown by the above mentioned triples. In all other cases the function value is equal to 0. We can now formalize the above mentioned or-relation by making use of the or-operator or the or-function. We say that f = 1 if

a'b'c + a'bc + ab'c' + abc = 1.

NOTATIONS FOR MINTERM FUNCTION

- Function: $f = \bar{a}\bar{b}c+\bar{a}bc+a\bar{b}\bar{c}+abc$
- Sum of minterms form = standard normal form:

$$f = m_1+m_3+m_4+m_7$$

$$f = \sum(1,3,4,7)$$

-4.33-

This leads to the expression shown in figure 4.33 that describes the function f. We see that f is given by an or-relation, a sum of a number of terms that are all minterms. Such an expression is called the *sum-of-minterms* form, or the standard normal form. Previously we have seen that in order to write a minterm form we do not need to write the complete algebraic expression, but that we also can refer to this minterm by the letter m with an index. The sum of minterms from shown in figure 4.33 (for f) can be denoted as $m_1 + m_3 + m_4 + m_7$. We frequently denote the sum of minterm from by making use of the sum symbol. This is shown in figure 4.33.

EXAMPLE - STANDARD NORMAL FORM

- Function: $f = \sum (0,1,4,5,10,11,12)$
- $f = m_0+m_1+m_4+m_5+m_{10}+m_{11}+m_{12}$
- $f = \bar{a}\bar{b}\bar{c}\bar{d}+\bar{a}\bar{b}\bar{c}d+\bar{a}b\bar{c}\bar{d}+\bar{a}b\bar{c}d$
  $+a\bar{b}c\bar{d}+a\bar{b}cd+ab\bar{c}\bar{d}$
- f = 1 for quadruples
  0000, 0001, 0100, 0101, 1010, 1011, 1100
- Truth table

a b c d	f	a b c d	f
0 0 0 0	1	1 0 0 0	0
0 0 0 1	1	1 0 0 1	0
0 0 1 0	0	1 0 1 0	1
0 0 1 1	0	1 0 1 1	1
0 1 0 0	1	1 1 0 0	1
0 1 0 1	1	1 1 0 1	0
0 1 1 0	0	1 1 1 0	0
0 1 1 1	0	1 1 1 1	0

-4.34-

Consider the following example. In figure 4.34 the function f is defined as the sum of the minterms 0, 1, 4, 5, 10, 11, and 12. We can denote this by making use of the sum or Σ symbol and relate the decimal numbers with the wanted minterms, or by using the notation $m_0 + m_1 + ... + m_{12}$. Both notational forms are equivalent. In both cases we want to express that the function f is a sum of the minterms shown in the third expression of figure 4.34. This is the only algebraic expression that determines the behaviour of the function f clearly. We can convert this expression to a truth table by considering the function f as being equal to 1 only when the combinations of the variables' values results in one of its minterms being equal to 1. Thus, f = 1 if the values of the inputs are equal to the quadruples related to the minterms. These quadruples are shown in figure 4.34. From these quadruples follows the given truth table. Notice that there is a very direct relation between the decimal numbers used at the beginning of this example to express the minterms, and the truth table. Indeed, the decimal numbers can be translated into their binary equivalent, yielding the n-tuples resulting in a function value equal to 1.

REMARKS - STANDARD NORMAL FORM

- Any switching function has a standard normal form
- Any switching function has its own unique standard normal form
- There are 2^nminterms with n variables
  - 1 function with sum of 0 minterms
  - $\binom{2^n}{1}$ functions with sum of 1 minterm
  - $\binom{2^n}{2}$ functions with sum of 2 minterms

  ⋮
- In total there are:
  $1+\binom{2^n}{1}+\binom{2^n}{2}+\cdots+\binom{2^n}{2^n} = (1+1)^{2^n} = 2^{2^n}$
  functions of n variables

-4.35-

The decimal number 0 produces the n-tuple 0000, in this case f is equal to 1. The decimal number 5 produces the n-tuple 0101, also in this case f is equal to 1. And finally, the decimal number 12 produces the n-tuple 1100 and f is also equal to 1. We see that the sum of minterms form and the truth table of an arbitrary function are closely related: they constitute the same type of specification.

We shall make some remarks about the sum of minterms form or the standard normal form (see figure 4.35). First we notice that each switching function can be written as a sum-of-minterms form, i.e. each switching function has a standard normal from. Secondly, it is valid that each function has its own *unique* standard normal form. Better said: two functions with the same standard normal form are identical.

Thirdly we shall consider the number of different functions with n variables. From the previous remark we know that this number is equal to the number of different standard normal forms. Previously we have seen that we can form 2^n different terms from n variables. A minterm may or may not relate to a sum of minterms. We say also that the minterm belongs or does not belong to a function.

NUMBER OF FUNCTIONS

# variables	# functions
1	4
2	16
3	256
4	65536
5	4.294.967.295
6	$\approx 1.8 \times 10^{19}$

- 0 variables ⟶ 2 functions

$f = 0 \ , \ f = 1$

- 1 variable ⟶ 4 functions

$f(a) = 0$

$f(a) = \bar{a}$

$f(a) = a$

$f(a) = 1$

-4.36-

NUMBER OF FUNCTIONS

- 2 variables ⟶ 16 functions

	ab	$a\bar{b}$	$\bar{a}b$	$\bar{a}\bar{b}$	function
f_0	0	0	0	0	$=0$
f_1	0	0	0	1	$=\bar{a}\bar{b}$ nor
f_2	0	0	1	0	$=\bar{a}b$ inhibit $a<b$
f_4	0	1	0	0	$=a\bar{b}$ inhibit $b<a$
f_8	1	0	0	0	$=ab$ and
f_3	0	0	1	1	$=\bar{a}$ inverse
f_5	0	1	0	1	$=\bar{b}$ inverse
f_6	0	1	1	0	$=a\bar{b}+\bar{a}b$ exclusive or
f_9	1	0	0	1	$=ab+\bar{a}\bar{b}$ equivalent
f_{10}	1	0	1	0	$=b$
f_{12}	1	1	0	0	$=a$
f_7	0	1	1	1	$=\bar{a}+\bar{b}$ nand
f_{11}	1	0	1	1	$=\bar{a}+b$ $a \leqslant b$
f_{13}	1	1	0	1	$=a+\bar{b}$ $b \leqslant a$
f_{14}	1	1	1	0	$=a+b$ or
f_{15}	1	1	1	1	$=1$

minterms

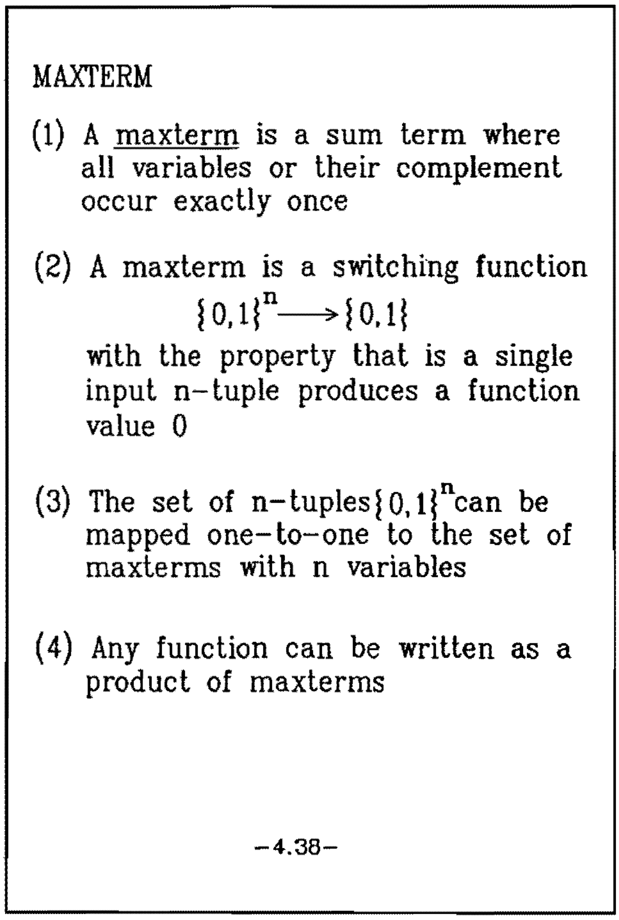
-4.37-

With 2^n minterms we can:

- form 1 function, described by a sum of 0 minterms.
- form (2^n over 1) functions, described by the sum of 1 minterm.
- form (2^n over 2) functions, described by the sum of 2 minterms etc.

If we add these numbers we get the total number of functions of n variables (see figure 4.35). What this means for the number of functions is shown in figure 4.36. For functions with 5 or more variables the number of different functions is gigantically large. Furthermore, figure 4.36 shows all functions of 0 variables and 1 variable. We must also consider that "the constant function with a value equal to 0 or 1", is a function. Figure 4.37 shows an overview of all functions with two variables. These functions are composed by making a selection from the 4 mentioned minterms. By using a 1 to denote that a specific minterm belongs to a function, and a 0 to denote that it does not, we can attach a decimal number to the different functions.

This is shown in figure 4.37. One of the functions we know is the function f_8; the and-function. The functions f_3 and f_5 are the inverse functions. The function f_{14} is the or-function. Later we shall also get acquainted with the function f_1, the *nor,* the function f_7, the *nand,* and the function f_6, the *exclusive-or.*
It will be clear to you that we did not make tables of all functions of 3 or more variables.



MAXTERM

(1) A <u>maxterm</u> is a sum term where all variables or their complement occur exactly once

(2) A maxterm is a switching function

$$\{0,1\}^n \longrightarrow \{0,1\}$$

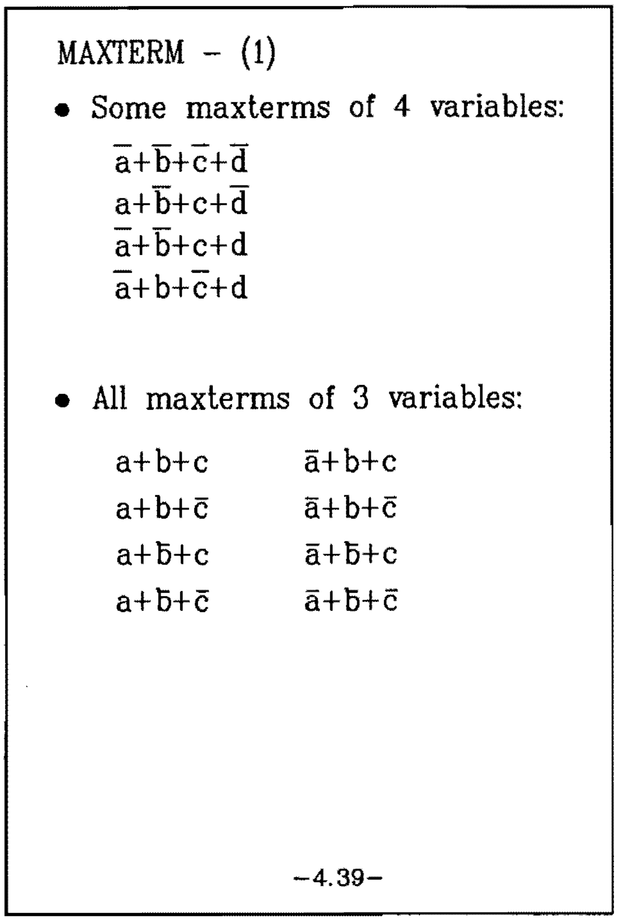
with the property that is a single input n–tuple produces a function value 0

(3) The set of n–tuples $\{0,1\}^n$ can be mapped one–to–one to the set of maxterms with n variables

(4) Any function can be written as a product of maxterms

–4.38–

We have seen that there exists a sum of minterms form for each function. A sum-of-minterms form is a special case of the sum of products form. Furthermore, there exists a product of sums form for each function. In this case each of the sum terms is a *maxterm*. Figure 4.38 shows a definition of a maxterm.



MAXTERM – (1)

- Some maxterms of 4 variables:

  $\bar{a}+\bar{b}+\bar{c}+\bar{d}$
  $a+\bar{b}+c+\bar{d}$
  $\bar{a}+\bar{b}+c+d$
  $\bar{a}+b+\bar{c}+d$

- All maxterms of 3 variables:

$a+b+c$	$\bar{a}+b+c$
$a+b+\bar{c}$	$\bar{a}+b+\bar{c}$
$a+\bar{b}+c$	$\bar{a}+\bar{b}+c$
$a+\bar{b}+\bar{c}$	$\bar{a}+\bar{b}+\bar{c}$

–4.39–

Figure 4.39 shows some maxterms that are composed of four variables; it also shows all maxterms composed of three variables. Analoguously to our treatment of minterms, we can now consider a maxterm as a switching function that maps a binary n-tuple to a single binary variable. The maxterm has the special characteristic that there is exactly one n-tuple for which the function value is 0. For all other n-tuples the maxterm's value is equal to 1.

MAXTERM - (2)

- A maxterm is a sumterm; hence it is a switching function
- $f = x_0+\overline{x}_1+....+x_i+...+\overline{x}_j+...+\overline{x}_{n-1}+x_n$
  is only equal to 0
  if all $x_i=0$ and all $\overline{x}_j=0$
  in other words if all
  $x_i=0$ and all $x_j=1$
- this defines precisely a single n-tuple

MAXTERM - (3)

- Construction of n-tuple:
  - replace all x_i by 0
    and
  - replace all $\overline{x}_j$ by 1
- Construction of maxterm:
  - Replace each 0 by the corresponding x_i
  - Replace each 1 by the corresponding $\overline{x}_j$

-4.40-

Figure 4.40 illustrates this further. We see that a maxterm is only equal to 0 if all present normal variables are equal to 0 and all the present inverted variables are equal to 1. Herewith we have given all possible variables a value; we have thus selected exactly one n-tuple from the set of possible n-tuples. For all the other n-tuples there is at least one normal variable equal to 1, or there is at least one inverted variable equal to 0. In all cases we get (for the maxterm) the value "1 or some-other-value", which is always equal to 1.

Once more, the set of n-tuples can be mapped one-to-one to the set of maxterms with n variables (see figure 4.38). Figure 4.40 shows how the n-tuples could be obtained from the maxterms, and also how the maxterms can be defined in terms of the n-tuples. We notice that exactly one binary n-tuple belongs to each maxterm, and vice versa.

EXAMPLE - CONSTRUCTION OF N-TUPLE

- 4 variables

$1011 \longleftrightarrow \overline{a}+b+\overline{c}+\overline{d}=0$ for $(a,b,c,d)=(1,0,1,1)$
$0010 \longleftrightarrow a+b+\overline{c}+d=0$ for $(a,b,c,d)=(0,0,1,0)$
$1010 \longleftrightarrow \overline{a}+b+\overline{c}+d=0$ for $(a,b,c,d)=(1,0,1,0)$

- With

n-tuple → binary number →
→ decimal number

$a+\overline{b}+\overline{c}+d \longleftrightarrow 0110 \longleftrightarrow M_6$
$\overline{a}+b+\overline{c}+d+e \longleftrightarrow 10100 \longleftrightarrow M_{20}$
$a+b+c+d \longleftrightarrow 0000 \longleftrightarrow M_0$
$\overline{a}+\overline{b}+\overline{c}+\overline{d}+\overline{e} \longleftrightarrow 11111 \longleftrightarrow M_{31}$

-4.41-

Figure 4.41 shows some examples of this relation between n-tuples and maxterms. Notice specially that a 1 in an n-tuple corresponds to the inversion of a variable; a 0 in an n-tuple corresponds to a non-inverted variable. For minterms the opposite holds. Also now we can consider, in the case of maxterms, the related n-tuple as a binary representation of a decimal number. We can refer to the maxterm with this decimal number. To distinguish it from a minterm we use the capital letter M with an index. See figure 4.41 for a number of examples. Compare this figure with figure 4.31 where similar cases of minterms are shown.

EXAMPLE – DERIVATION: PRODUCT OF MAXTERMS

- Truth table

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

- f is 0 for triples 000, 010, 101, 110
- f is 0 if maxterm $a+b+c$ or $a+\bar{b}+c$ or $\bar{a}+b+\bar{c}$ or $\bar{a}+\bar{b}+c$ is 0
- f is 1 if maxterm $a+b+c$ <u>and</u> $a+\bar{b}+c$ <u>and</u> $\bar{a}+b+\bar{c}$ <u>and</u> $\bar{a}+\bar{b}+c$ <u>all are 1</u>
- f is 1 if $(a+b+c)\cdot(a+\bar{b}+c)\cdot(\bar{a}+b+\bar{c})\cdot(\bar{a}+\bar{b}+c)$ is 1
- $f = (a+b+c)\cdot(a+\bar{b}+c)\cdot(\bar{a}+b+\bar{c})\cdot(\bar{a}+\bar{b}+c)$

$$= M_0 \cdot M_2 \cdot M_5 \cdot M_6$$

$$= \Pi(0,2,5,6)$$

= product of maxterm form

–4.42–

Finally we notice (in figure 4.38) that each function can be written as a product of maxterms. The example in figure 4.42 shows how we can derive this product-of-maxterms form.

First we note the triples yielding a function value 0. Each triple corresponds to a maxterm that is only equal to 0 for this combination of variables. The function f is equal to 0 if a single of these maxterms is equal to 0. In all other cases the function value is equal to 1. Specially, the function value is equal to 1 if all the (found) maxterms are equal to 1: if each of the maxterms has the value 1, the product of these maxterms is also equal to 1. Thus, it is valid that f = 1 if the product of the maxterms is equal to 1, or f = $M_0 \cdot M_2 \cdot M_5 \cdot M_6$. We are considering a *product-of-maxterms* form. It is advised to compare the derivation of this product-of-maxterms form with the corresponding derivation of the *sum-of-minterms* form.

MINTERM VS. MAXTERM FORM

- Consider

$$f = m_i$$

– f is 1 for the n-tuple corresponding to the binary value of i

- Consequently

$$\bar{f} = \overline{m_i}$$

– $\bar{f}$ is 0 for the n-tuple corresponding to the binary value of i

- Consequently

$$\overline{m_i} = M_i$$

and

$$\overline{M_i} = m_i$$

- The complement of a minterm with binary value i is equal to the maxterm with the same number

–4.43–

For the same function we can describe the sum-of-minterms form as a product-of-maxterms form. These two notational forms must be directly related to each other. In figure 4.43 we have a function given as a sum of one minterm, or as the minterm m_i. We know that this function only has the value 1 for one possible combination of input values. The function is 1 for that n-tuple with a binary value is equal to i. Actually, the inverse (m_i') of the function is equal to 0 for that n-tuple. Previously we have seen that this n-tuple defines the maxterm M_i. The conclusion is then that the inverse of a minterm with a index number i, is equal to the maxterm with the same index. This conclusion is in agreement with De Morgan's laws.

Problem 4.2 (4.1)

Show, with the help of De Morgan's laws, that inverse of the minterm m_{10} of five variables is equal to the maxterm M_{10}.

Problem 4.3 (4.3)

Show whether

- *the sum-of-minterms forms and*
- *the product-of-maxterms forms*

are identical for functions f_1 and f_2 in the following cases:

a. $f_1(x,y) = \overline{(x\overline{y}+\overline{x}y)}$

   $f_2(x,y) = xy+\overline{x}\overline{y}$

b. $f_1(w,x,y,z) = w\overline{xz}+wx+\overline{w}x\overline{y}$

   $f_2(w,x,y,z) = (x+\overline{z})(w+x)(w+\overline{y})$

c. $f_1(x,y,z) = \overline{(\overline{y}+\overline{x}z)}$ *and* $f_2(x,y,z) = xyz+\overline{x}y\overline{z}+$

d. $f_1(x,y,z) = (x+\overline{y})(x+y+z)(x+y+\overline{z})$

   $f_2(x,y,z) = x$

INVERSION OF FUNCTION

- Function: $f=\sum(\text{minterms})$
- Inverse: $\bar{f}=\sum(\text{remaining minterms})$

- Example: $f=\sum(1,4,5,7)$
- Inverse: $\bar{f}=\sum(0,2,3,6)$

according to the truth table:

a	b	c	f	$\bar{f}$
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	0	1
1	1	1	1	0

–4.44–

RELATION MINTERMS – MAXTERMS

Function: $f=\sum(\text{minterms})$

Inverse: $\bar{f}=\sum(\text{remaining minterms})$

Accordingly:

$$f=\bar{\bar{f}}=\overline{\sum(\text{remaining minterms})}$$

$$\overset{\text{De Morgan}}{=}\prod(\overline{\text{remaining minterms}})$$

$$=\prod(\text{remaining maxterms})$$

Example 1 (continued from fig. 4.44):

$$f(a,b,c)=\bar{\bar{f}}=\overline{\sum(0,2,3,6)}=\overline{m_0+m_2+m_3+m_6}$$

$$=\bar{m}_0\cdot\bar{m}_2\cdot\bar{m}_3\cdot\bar{m}_6$$

$$=M_0\cdot M_2\cdot M_3\cdot M_6 \quad =\prod(0,2,3,6)$$

Example 2:

$$f(a,b,c)=\sum(0,1,2,6,7)=\prod(3,4,5)$$

–4.45–

Consider now a function f that is given as a sum of minterms (figure 4.44). The inverse of f is equal to 0 when f itself has the value 1, and the inverse of f is equal to 1 when f itself has the value 0. This latter case is true for the minterms that do not belong to f; these are not part of the sum of minterms for f. In other words the inverse of f is 1 for the minterms that are not a part of the sum of minterms for f. These we call the "other minterms". The inverse of f is determined by the sum of these other minterms. Figure 4.44 explains this with the help of an example. The function f is given as the sum of the minterms 1, 4, 5 and 7. Therefore, the inverse of f must be equal to the sum of the other minterms. These are the minterms 0, 2, 3 and 6. The truth table shows this unambiguously. The functions f and f' are complements to each other. When f = 1, f' = 0 and vice versa.

We shall now take a further step (figure 4.45). We know that if a function is given by the sum of a number of minterms, the inverse function is given by the sum of the other minterms. The involution characteristic (see theorem 6 in figure 4.15) states that the inverse of the inverse of f is equal to f. Thus f = f'' is the inverse of the sum of the other minterms. With the theorem of De Morgan, the inverse of a sum is transformed into a product of the inverses of the other minterms. The inverse of a minterm is the maxterm with the same index number. Finally we find that f can also be written as the product of the "other maxterms".

To summarize: a function f can be written as a sum of a number of minterms, and also as the product of the other maxterms. Let us, for further explanation, complete the example that we started in figure 4.44 (see figure 4.45). Previously we have derived that the inverse of f can be written as the sum of the other minterms 0, 2, 3 and 6. Thus f itself is equal to the inverse of the sum of these minterms, or according to De Morgan, it is equal to the product of the inverse of these minterms. Actually, the inverse of the minterm m_0 is the maxterm M_0, and the inverse of m_2 is maxterm M_2 etc. Finally we find that f can also be written as the product of the maxterms 0, 2, 3 and 6. These were exactly the missing numbers in the sum of the minterms form of f. Another example (see figure 4.45) shows f(a,b,c) to be given by the sum of minterms 0, 1, 2, 6 and 7. Thus f is also equal to the product of the maxterms 3, 4 and 5.

Problem 4.4 (4.4)

Consider the function T of 4 variables:

$T(w,x,y,z) = \bar{w}\bar{x}\bar{y}+xz+\bar{w}\bar{y}z$

The following holds for the numeric representation of the minterms:
the variable w has the highest weight; z has the lowest one.

a. *Write the function in numerical form as a sum of minterms, i.e. as:* $T = \sum(\ldots)$.

b. *Write the function in numerical form as a product of maxterms.*

c. *Consider T' being the inverted function of T: repeat questions a. and b. for T'.*

d. *Do the same for T_d being the dual function of t (see survey page 4.7)*

Remark: you need hardly to do any computation work for problems b, c and d.

Problem 4.5/4.6/4.7 (4.5/4.6/4.7)

Work out the questions in problem 4.4 a - d also for the following functions:

4.5. $T(w,x,y,z) = (\bar{w}+\bar{x}+y+z)(x+\bar{y}+z)(w+x+\bar{y})(w+\bar{y}+z)(w+y+\bar{z})$

4.6. $T(w,x,y,z) = \bar{w}x(xz+y\bar{z})+z(w+x)(\bar{w}+\bar{x})$

4.7. $T(w,x,y,z) = (wz+y)(\bar{w}+\bar{x}+\bar{y}+\bar{z})+wxyz$

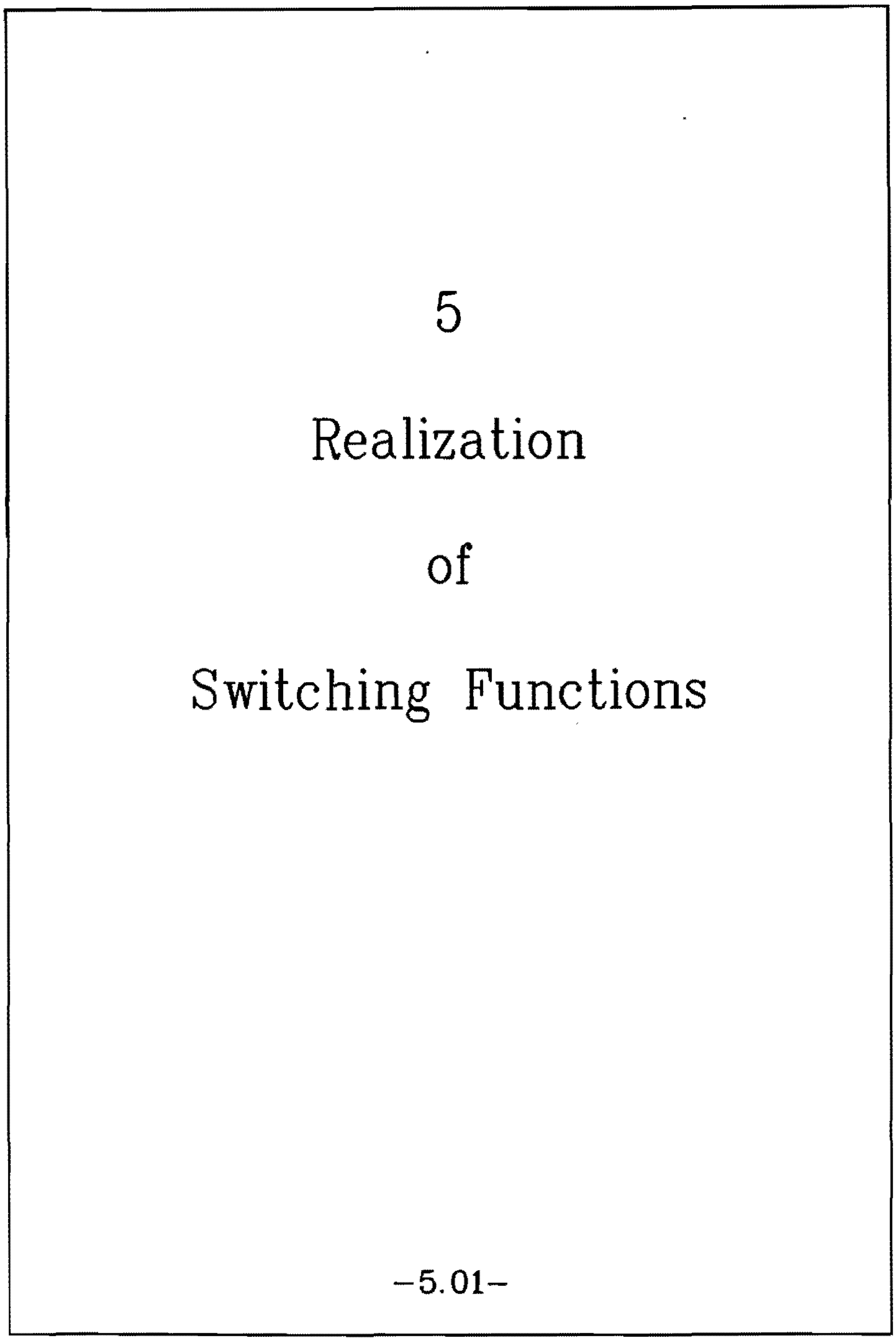
4.4 Summary

In this chapter we have become acquainted with switching algebra and its relation to binary systems. We have applied switching algebra for the behavioural description of binary systems, because the exclusive use of truth tables would have limited our possibilities. Switching algebra is related to a set P of the two elements 0 and 1, and defines two operations, the sum and the product, that are closed over P. Furthermore, we have given five postulates from which we have derived seven theorems. Postulates and theorems enable us to evaluate expressions with "and" and "or"-operations, to process and to simplify them. During this course we shall make regular use of these theorems and postulates; you would do yourself a favour by knowing them.

An aspect of this chapter is that, in any case, we could use electronic devices to make circuits to implement the function of the sum and product operators, producing high and low voltages. We are considering the and-gate and the or-gate.

In the following chapter we shall see how to realize general sum-of-products forms from these and and or-gates.

The application of an algebra enables us to reason more formally about the sum and product relations between variables and their implication for combinational binary systems. We have seen that there exist two standard normal forms closely related to the truth table. The first one is the sum-of-minterms form that is also called the *disjunctive normal form.* The second normal from is the product-of-maxterms form, which is also called the *conjunctive normal form.* Later we shall frequently use the abbreviation SOM (Sum-Of-Minterms). We have seen that both standard normal forms exist for all functions. Moreover, we have learned how to derive one standard normal form from the other, and we know the relation between both forms and the truth table. In the following chapter we shall make use of this knowledge to find realizations of combinational binary functions.



5

Realization

of

Switching Functions

−5.01−

5 Realization of switching functions

In this chapter we shall discuss the realization of combinational binary systems being described by switching functions, by mapping them to standard components. First we shall consider the realization of switching functions described by the sum-of-minterms form. Then we shall consider the more general sum-of-products form and its realization possibilities.

Furthermore, we shall see that we can realize any switching function with only one type of gate. In the previous chapters we have described the behaviour of the multiplexer, and we have seen the possible systems and architectures for it. In this chapter we shall show that the multiplexer can be used as a universal building block for the realization of switching functions. Finally we shall discuss a new type of building block for easy realization of specific functions.

Realization

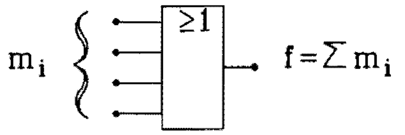
of a

Sum of Minterms

-5.02-

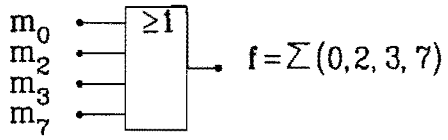
SUM OF MINTERMS: AND/OR GATES (1)

f = sum of minterms = Σm_i
= or-function of minterms



Example

$f=\Sigma(0,2,3,7) = m_0+m_2+m_3+m_7$



-5.03-

5.1 Realization of a sum of minterms

In the previous chapter we have seen that we can describe any switching function as a sum of minterms.

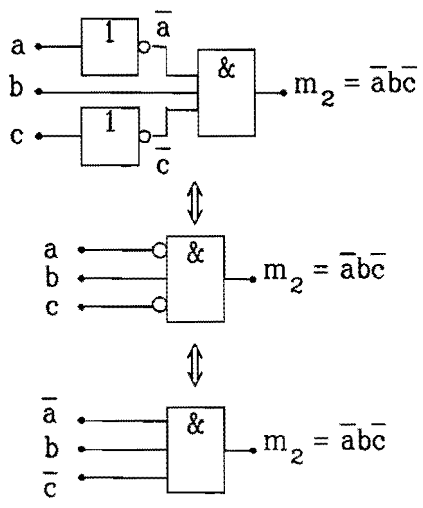
We can also say: f is given by an or-function of a number of minterms (see figure 5.03). For the realization of such a function we can use an or-gate with as many inputs as there are minterms in the function. For each of the inputs we must realize a new function, equal to one of the minterms of the function.

We shall take as an example a function f which is given by the sum of minterms 0, 2, 3 and 7. This function can be realized by an or-gate with four inputs, where on each input we must realize one of the minterms m_0, m_2, m_3 or m_7. We know that if one of these terms is equal to 1 then the output of the or-gate will be also equal to 1.

The realization of a function as a sum of minterms is now reduced to the realization of a number of minterms. We have previously defined a minterm as a product term, where all variables occur either in normal or complemented form.

SUM OF MINTERMS: AND/OR-GATES (2)

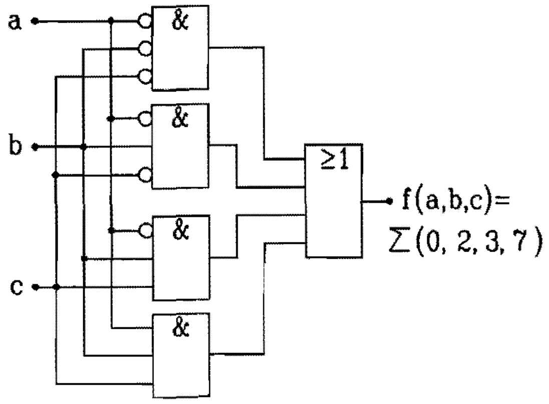
- Minterm: and-function of variables and their complement
- Realization: and-gate + inverters
- Example: $m_2 = \bar{a}b\bar{c}$



-5.04-

SUM OF MINTERMS: AND/OR-GATES (3)

- Sum of minterms: 2-layer and/or realization
- Example:
  $f = \Sigma(0, 2, 3, 7)$



-5.05-

We can also say (see figure 5.04) that a minterm (an and-function) is composed of variables and/or inverted variables. Thus, a minterm can be realized by an and-gate and a number of inverters. The and-gate must have as many inputs as there are variables. Figure 5.04 shows, as an example, the realization of the minterm m_2. Notice that it is necessary to first invert the variables a and c, before their values are passed to the and-gate. In chapter 4 we have already stated that it is not always necessary to draw an inverter as a separate symbol. We can denote the inverter function by drawing a small circle at the related input of the and-gate. We arrive at the second schematic in figure 5.04. Sometimes we also do not worry about the realization of the inverted a and c, and we show which specific variables need to be inverted at the inputs of the and-gate. This is shown is the third schematic in figure 5.04. Thus, these three schematics represent alternative realizations of the same minterm. The drawing method to be used depends on the environment.

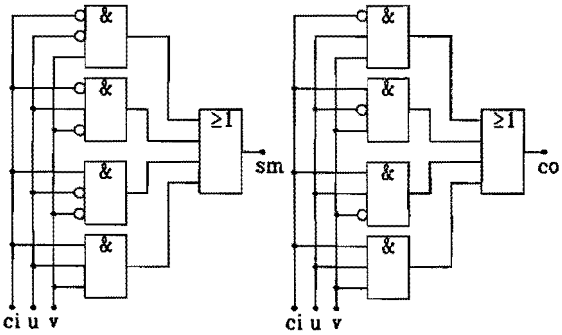
The combination of these and the previous figure shows a method for the realization of a sum-of-minterms form. Figure 5.05 shows an example. Notice that, if we do not consider the inverters, the realization form is a 2-layer tree structure. In layer one we have the or-gate, layer two is composed of a number of and-gates. Therefore, we are considering a 2-layer and-or realization. As an example we have chosen a function which is given by the sum of minterms 0, 2, 3 and 7. We see that we need 4 and-gates for the realization of the minterms. The topmost and-gate realizes the minterm m_0, the following and-gate realizes m_2. The following one m_3, and finally the last and-gate realizes m_7. Verify this for yourself. Notice that now at most one output of the and-gates is equal to 1 and the rest of the outputs are equal to 0. The four and-gates' outputs are combined in the or-gate, where the function value is available at its output.

2-LAYER AND/OR - EXAMPLE: ADD2

- Truth table

ci	u	v	sm	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- $sm = m_1+m_2+m_4+m_7 = \sum(1,2,4,7)$
- $co = m_3+m_5+m_6+m_7 = \sum(3,5,6,7)$



-5.07-

REALIZATION WITH FIXED ARCHITECTURE

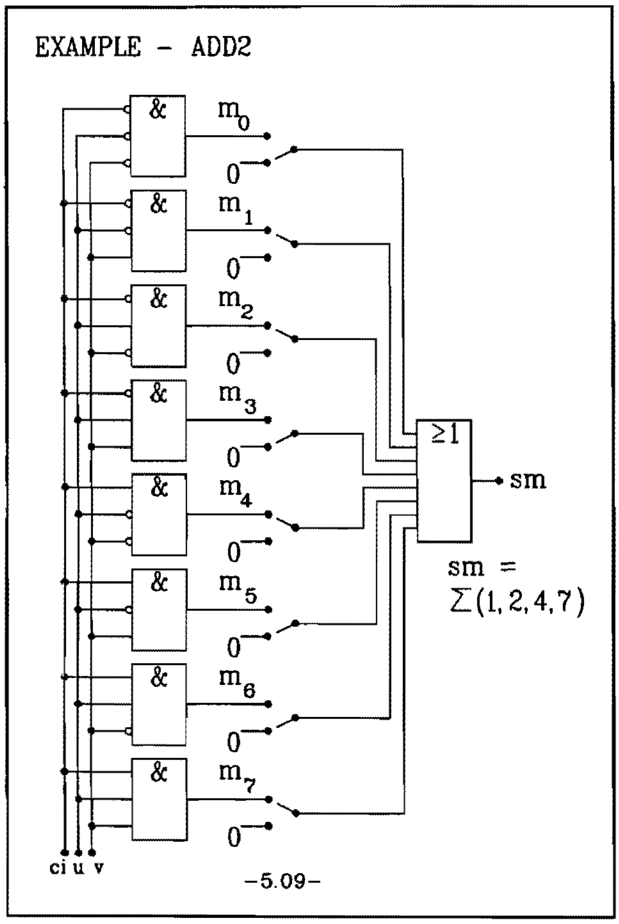
- Architecture: 2-layer and/or

- Universal building block:
  - realize all minterms of n variables with 2^n and-gates
  - connect the minterms with an or-gate

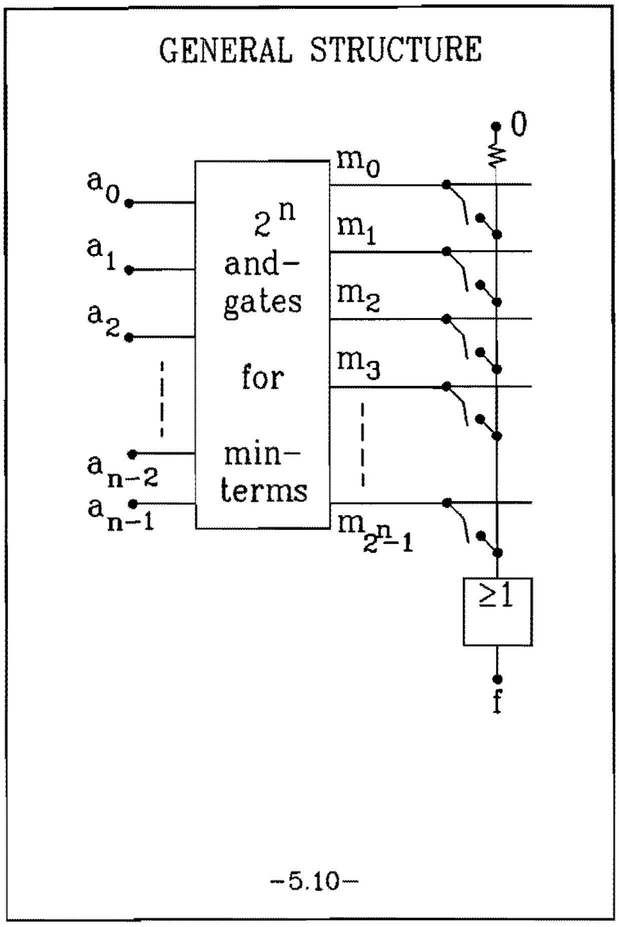
-5.08-

Earlier (figure 4.05) we constructed the truth table of our ADD2 module. This truth table has been repeated in figure 5.07. We shall realize this module using and-gates and or-gates. We state first that the sum output, sm, is given by the sum of the minterms 1, 2, 4 and 7 and that the carry output, co, is given by the sum of the minterms 3, 5, 6 and 7. For each of the two switching functions we get a 2-layer and-or realization. Verify that the named minterms can be realized using and-gates. Notice also that both functions sm and co have the minterm m_7 in common. We could have realized this minterm just once, connecting its output to both or-gates. This would save an and-gate. We shall return to this later.

We have now shown that the realization of a sum of minterms occurs according to a fixed pattern; it is realized according to a fixed *architecture*. This is the two-layer and-or tree structure (see figure 5.08). Applying this general architecture we can create a universal building block used to realize any switching function of n variables. In this building blocks we first construct all minterms of n variables. Therefore, we need 2^n and-gates. Those and-gates corresponding to minterms belonging to the sum-of-minterms function, should be connected to the or-gate.



-5.09-


GENERAL STRUCTURE
a_0 a_1 a_2 a_{n-2} a_{n-1}
2^n and-gates for min-terms
m_0 m_1 m_2 m_3 m_{2^n-1}
0
≥1
f

-5.10-

As an example we show in figure 5.09 such a universal building block of three variables. Notice that we construct all minterms of three variables with the 8 and-gates, and that we connect the necessary minterms to the or-gate using switches. In figure 5.09 the switches are set such that the sm function of our ADD2 module is realized. Notice that the inputs of the or-gates that are not connected to an and-gate's output must be connected to a logic 0.

In figure 5.10 the idea of a universal building block is more generally worked out. The system's n-inputs a_0 to a_{n-1} are connected to a black box, where we assume it contains the 2^n and-gates for the construction of the minterms. These minterms are available at the right hand side of the black box, and are connected by means of a switch to an or-gate. Notice that only one minterm has the value 1, and all other minterms have the value 0.

Furthermore, we notice that we have drawn one input for the or-gate; in reality this gate has 2^n inputs. Figure 5.10 only shows a global schematic picture. We shall not deal with implementation details further in this course. What is meant is that by making a connection we can add a specific minterm to the (already constructed) sum of minterms at the output.

Since we have already constructed these 2^n minterms in the building block, is it a waste to use these minterms to only make one function. We can simply expand the number of outputs by adding a number of or-gates.

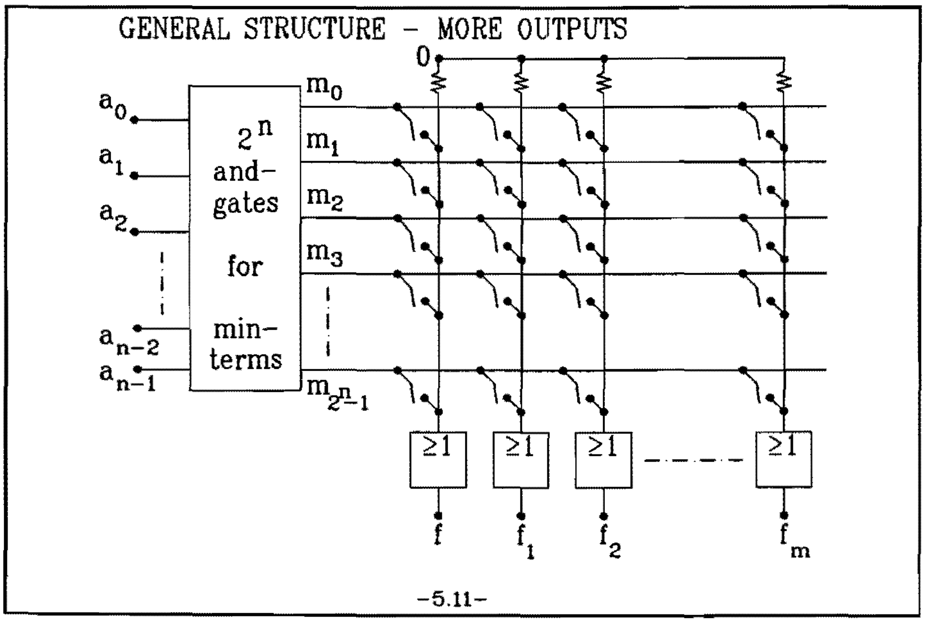
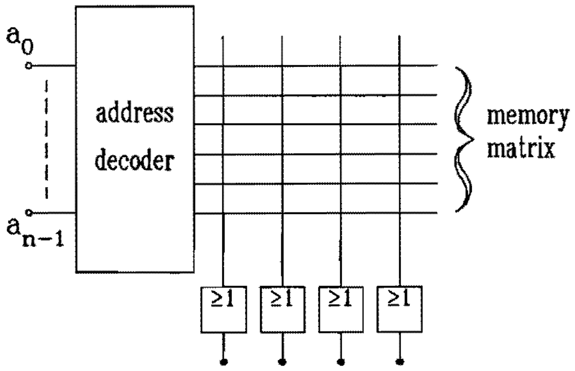


Figure 5.11 shows this. We can now use the same set of realized minterms to make several sum-of-minterms forms. Of course this is more economical than realizing all minterms for each function separately.

READ-ONLY MEMORY - ROM

- n inputs = address lines
- Address decoder: 1 out of 2^n minterms
- Memory words



-5.12-

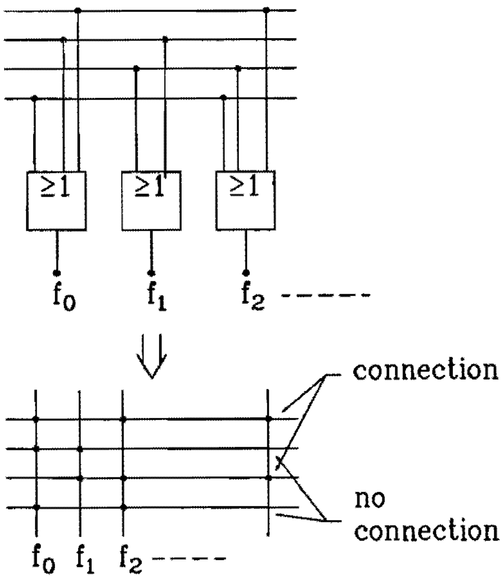
A universal building block such as the one just described exists in reality and is called a *Read Only Memory* (ROM), see figure 5.12. The reason for calling this building block a ROM is not important at this point. What is important, however, is the use of a number of different terms. First we call the n inputs of the building block *address lines*. These address lines go to an *address decoder*, its operation being the same as a minterm generator. The address decoder has 2^n outputs; only one output is equal to one, corresponding to the selection of 1 out of 2^n minterms.

These outputs go to the memory matrix, which in fact is nothing more that the set of switches in our universal building block. One horizontal row of a set of switches is called a *word,* and an individual switch represents a bit in a word.

We say now that the address decoder selects a word from the memory matrix by means of its output that is equal to 1. The individual bits in that word become the values on the corresponding outputs. A 0 for such a bit means that the corresponding output is a 0, A 1 means that the corresponding output is a 1.

CONNECTION MINTERMS/OR GATES

Way of drawing:



-5.13-

In figure 5.13 we shall further examine the way minterms must be connected to or-gates. If we do so in larger circuits by drawing individual lines from the output of the corresponding minterm to the input of an or-gate, we shall quickly arrive at a complex situation due to the large number of lines. Also drawing open or closed switches can produce a lot of detailed drawing work, and could be a source of errors.

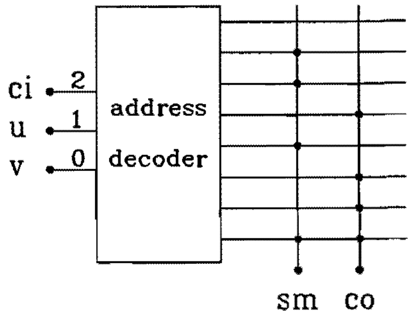
Figure 5.13 shows a frequently used method. A connection between a minterm and an or-gate is shown by placing a dot or circle at the intersection point in the memory matrix. So both schematics in figure 5.13 show the same circuit.

We shall discuss how to make these connections in a ROM. That is, we shall consider the *programming* of the ROM, which can be done in the following ways:

1. By using a specific mask while manufacturing the IC. The contents are thus determined in the factory. This device is refered to as a ROM.
2. The contents of another device type can also be programmed by the user (in the field). Therefore, the user needs special equipment to program such devices. We are considering a *Programmable ROM* or a PROM.
3. We speak about an *Erasable PROM* or an EPROM if the user can erase the contents of the ROM after using it, and reprogram this EPROM later.

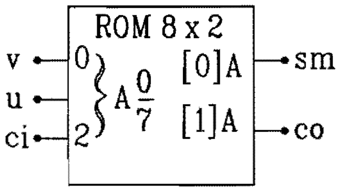
EXAMPLE - ROM

- ADD2 module



Truth table : fig. 5.07

- Standard IEC symbol



-5.14-

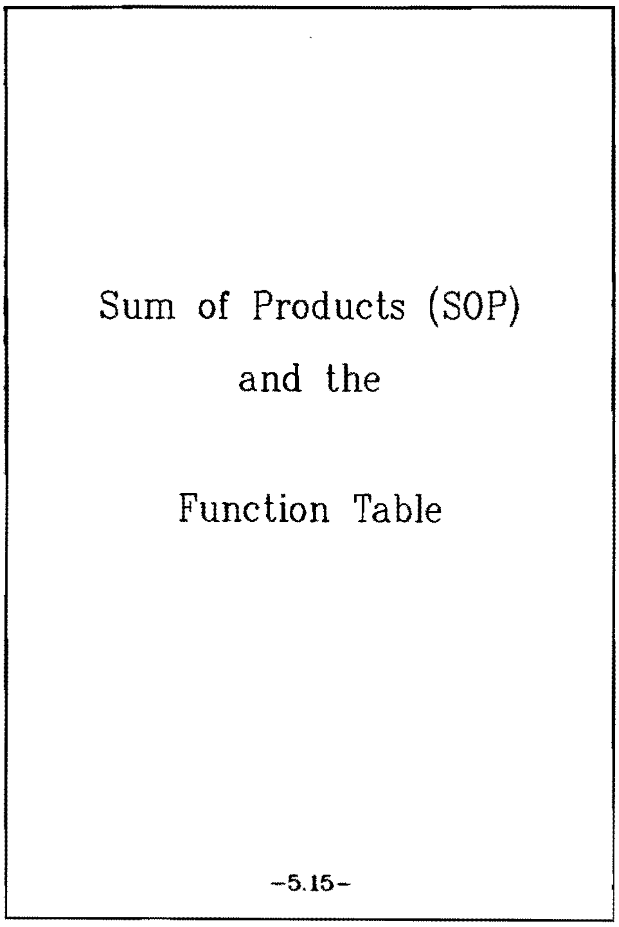
In figure 5.14 we have (as an example) realized our ADD2 module in a ROM with 8 memory words, i.e. 3 address inputs and 2 outputs. This is called an 8*2 ROM. During the implementation we have assumed that the input ci corresponds to the most significant address bit and the input v corresponds to the least significant address bit. Furthermore, we assume that the minterms m_0 to m_7 are at the outputs of the address decoders from top to bottom. If we compare the placing of the circles of the outputs sm and co with the truth table of figure 5.07 then we see a strong correspondence. Where there is a 1 in the truth table we get a circle.

Problem 5.1 (5.1)

In chapter 3 (figure 3.18) we gave a description of the COMP4 module. This module compares two numbers, with a value between 0 and 3, and gives as a result "greater", "equal" or "less". if we code this result as following: "greater" = 100, "equal" = 010 and "less" = 001. Give a ROM implementation of COMP4.

Make a suitable coding for the numbers on both inputs.

In figure 5.14 we have drawn the schematic of a ROM implementation of an ADD2 module, where we made use of the standard ROM symbol. It is clear (in this symbol) what the address lines are, where the least significant bit is, where the most significant bit is, and the address domain. Also the bit positions of the corresponding output in the ROM words are given with [0]A and [1]A. "A" indicates that the output depends on the A-inputs (Address lines). Notice that we have no information about the contents of the ROM. From the symbol alone we can not see what the function of this building block will be. Therefore, we need extra information. Schematics which are built with these symbols, show clearly how the system must be built, i.e. show the structure, but give no information about the *functionality* of the system, i.e. what the system must do. Therefore, a schematic is not a full system specification.

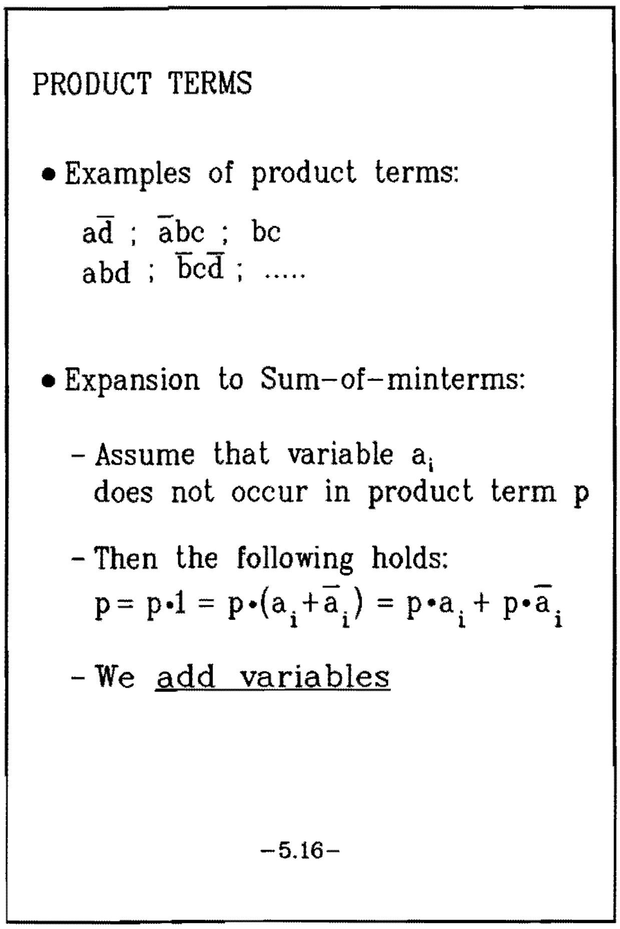


Sum of Products (SOP)

and the

Function Table

-5.15-



PRODUCT TERMS

- Examples of product terms:

  $a\bar{d}$; $\bar{a}bc$; bc
  abd ; $\bar{b}c\bar{d}$; .....

- Expansion to Sum-of-minterms:

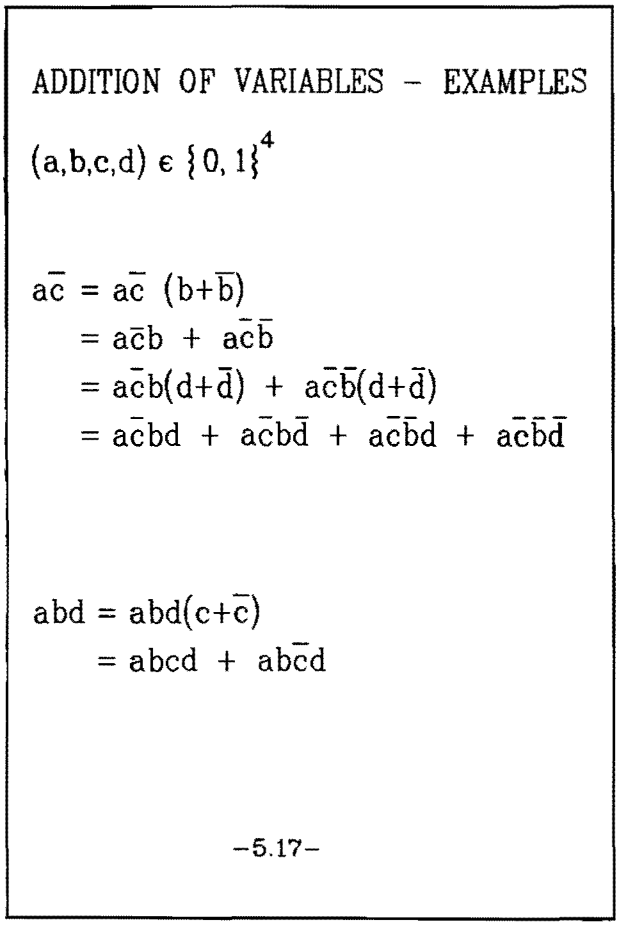
  - Assume that variable a_i does not occur in product term p
  - Then the following holds:
    $p = p\cdot 1 = p\cdot(a_i+\bar{a}_i) = p\cdot a_i + p\cdot\bar{a}_i$
  - We <u>add variables</u>

-5.16-

5.2 Sum Of Products (SOP) and the function table

We have seen that we can realize any function in its sum-of-minterms form. We have a building block, a ROM, that we only need to program. Currently EPROMs of 64k*8 ($1k=1024=2^{10}$) words are available. With such a ROM we can realize any 8 functions with 16 variables. Here a small problem arises. A function of 16 variables can contain a maximum of 65,536 minterms. It is questionable whether we can make error-free specifications of similar functions in a sum-of-minterms form. We are not restricted to only make use of a sum of minterms; we can also make use of the more general *product terms*.

In figure 5.16 we have described some product terms with 2 or more variables. Each product term can always be re-described as a sum of minterms. Indeed, let us assume that a variable a_i does not occur in a product term p. Figure 5.16 shows that this product term can be written as a sum of two new product terms: pa_i and pa_i'. The variable a_i occurs in both product terms. We have added a variable. We can go on with this process until all variables occur, in normal or inverted form, in each product term. Thus, we have arrived at a sum of minterms.



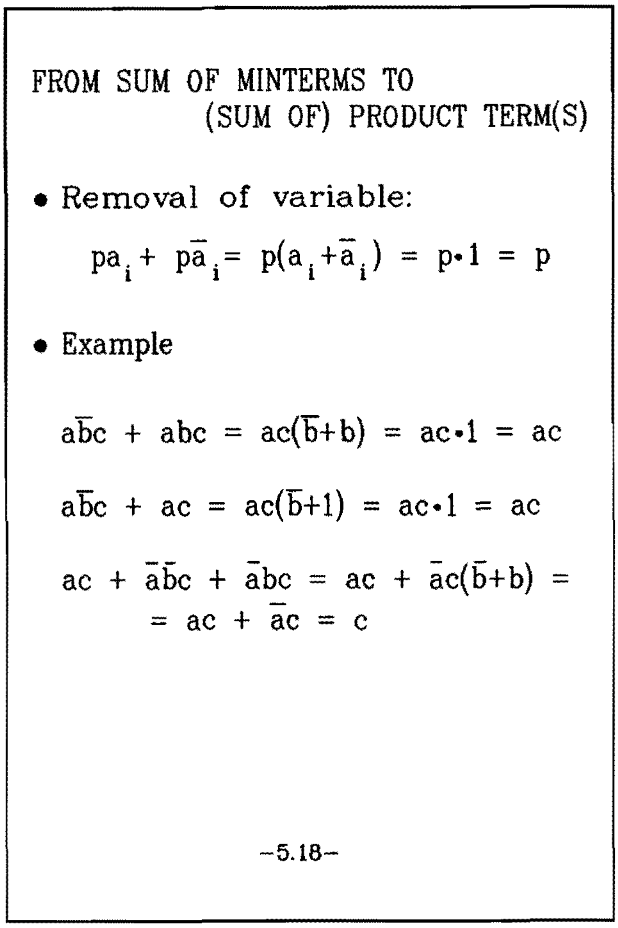
ADDITION OF VARIABLES – EXAMPLES

$(a,b,c,d) \in \{0,1\}^4$

$$
\begin{aligned}
a\bar{c} &= a\bar{c}\ (b+\bar{b}) \\
&= a\bar{c}b + a\bar{c}\bar{b} \\
&= a\bar{c}b(d+\bar{d}) + a\bar{c}\bar{b}(d+\bar{d}) \\
&= a\bar{c}bd + a\bar{c}b\bar{d} + a\bar{c}\bar{b}d + a\bar{c}\bar{b}\bar{d}
\end{aligned}
$$

$$
\begin{aligned}
abd &= abd(c+\bar{c}) \\
&= abcd + ab\bar{c}d
\end{aligned}
$$

–5.17–



FROM SUM OF MINTERMS TO
(SUM OF) PRODUCT TERM(S)

- Removal of variable:

$$pa_i + p\bar{a}_i = p(a_i + \bar{a}_i) = p \cdot 1 = p$$

- Example

$$a\bar{b}c + abc = ac(\bar{b}+b) = ac \cdot 1 = ac$$

$$a\bar{b}c + ac = ac(\bar{b}+1) = ac \cdot 1 = ac$$

$$
\begin{aligned}
ac + \bar{a}\bar{b}c + \bar{a}bc &= ac + \bar{a}c(\bar{b}+b) = \\
&= ac + \bar{a}c = c
\end{aligned}
$$

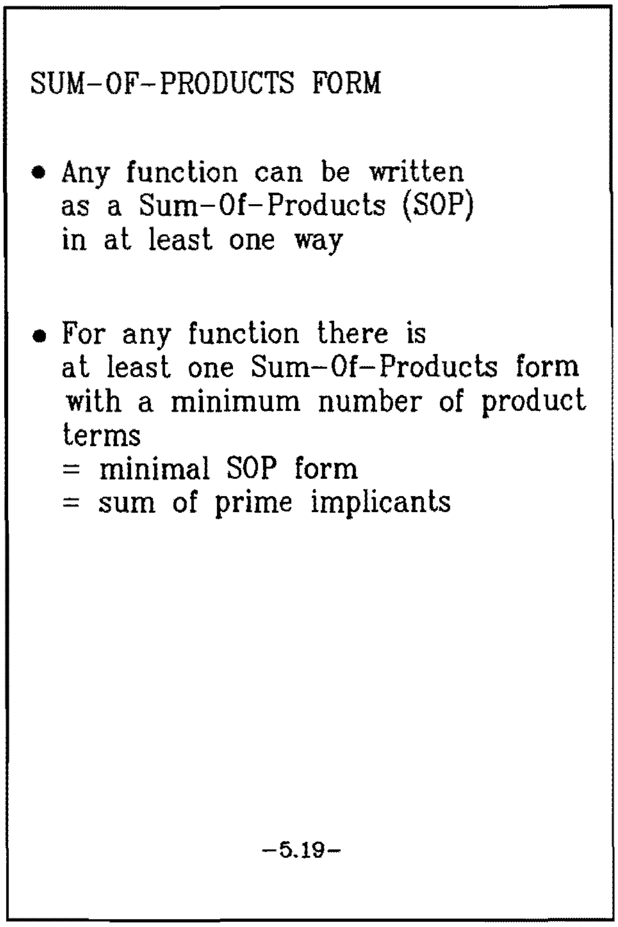
–5.18–

The first example in figure 5.17 shows how the variable b is added to the product term ac'. This results in two product terms; however, the variable d is still missing. Consequently, it is added to produce a sum of four minterms. In the second example variable c is missing; it is added to produce a sum of two minterms.

Problem (5.2)
Write the following two product terms as a sum of minterms

1: bd as a sum of minterms with 4 variables.
2: a as a sum of minterms with 3 variables.

Naturally, we can also do it the other way around. Figure 5.18 shows how we can go from a sum of minterms to a (sum of) product term(s). Suppose that we have 2 product terms that are identical except for 1 variable. This variable occurs in its normal form in one product term, and in its inverted form in the other product term. We can now combine both product terms into a new product term where the corresponding variable does not occur. In this way we eliminate a variable from the product terms. Figure 5.18 shows three examples. In the first example the variable b can be omitted from both product terms, and the product terms combine to a new product term. Example 2 shows us that if a variable is missing in one of the two product terms, we can still combine both product terms into a new product term, where the corresponding variable is eliminated. Finally, in the third example the variable b is deleted from the last two product terms, and then the two resulting product terms are combined to produce only one product term with one variable. We see that in all cases the number of product terms is decreased and that the number of variables per product term also decreases.



SUM-OF-PRODUCTS FORM

- Any function can be written as a Sum-Of-Products (SOP) in at least one way

- For any function there is at least one Sum-Of-Products form with a minimum number of product terms
  = minimal SOP form
  = sum of prime implicants

-5.19-

We now make the following assertion (figure 5.19):

> *Any function can be written in one or more ways as a Sum-Of-Products (SOP) form.*

Indeed, we have seen earlier that for any function there exists a sum-of-minterms form. This is a sum-of-products form. Thus, the assertion is correct. We can frequently eliminate variables from the minterms and combine the minterms to other product terms. In this way we arrive at other sum-of-products forms for the same function.

Frequently the number of product terms that we get is smaller than the number of minterms that we started with. If we master this combination of product terms and the elimination of variables, we can finally come up with a sum-of-products form where the number of product terms is minimal. That is, there is no sum-of-products form with a smaller number of product terms for the same function. We now state:

> *For each function there is at least one sum-of-products form with a minimum number of product terms. This form is called the **minimal sum-of-products** form.*

The proof of this statement lies beyond the scope of this course. We shall notice that this minimum occurs when all the product terms belong to the class of the so-called *prime implicants* of the function.

MINIMAL SOP FORM - ADVANTAGES

Minimal SOP realization has advantages above SOM realization

- Less product terms lead to
  - less and-gates
  - less or-gate inputs
- Less variables in a product term lead to less and-gate inputs
- Summary:

  less components + less wiring
  = less costs

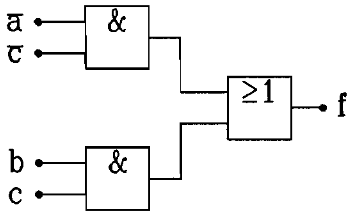
-5.20-

MINIMAL SOP FORM - EXAMPLE

- Function:

$$f = \ (0,2,3,7)$$
$$= \bar{a}\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}bc + abc$$
$$= \bar{a}\bar{c}(\bar{b}+b) + bc(\bar{a}+a)$$
$$= \bar{a}\bar{c} + bc$$

- Realization:



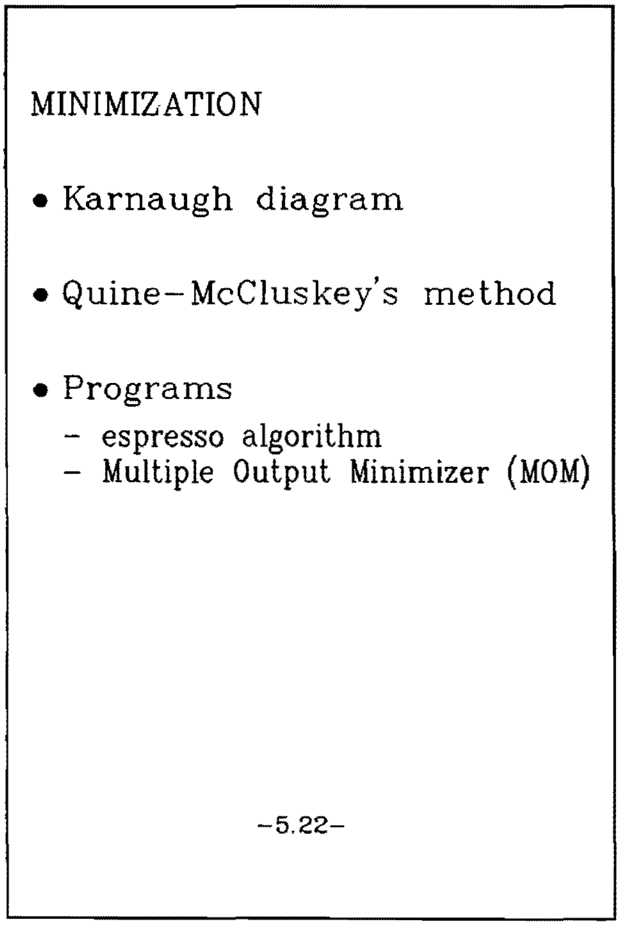
- Former realization (fig.5.05) :
  - 4 and-gates with 3 inputs
  - 1 or-gate with 4 inputs

-5.21-

Why is the sum-of-products forms so important? What is the advantage of the minimum sum-of-product form over the sum-of-minterms form? Figure 5.20 answers this question. First we shall notice that the sum-of-products form could be realized in the same way as a sum-of-minterms form. We have to deal with a sum form which, therefore, can be realized with an or-gate. The inputs of this or-gate are connected to the realizations of different product-terms.

These product-terms are only and-functions of variables, and thus can be realized with an and-gate. In this view, the sum-of-products form realization strongly resembles the sum of minterms realization. The minimum SOP form realization has as an advantage over the SOM form realization, that the number of product-terms to be realized is smaller than the number of minterms. A smaller number of product-terms means that we need less and-gates for the realizations. Thus, this is cost-saving. The use of less and-gates also means that the number of inputs of the or-gate can be smaller. Therefore, the realization of the or-gate will be cheaper. Furthermore, a product-term contains less variables than a minterm. This means that we not only need less and-gates but also that in general the and-gates could be realized with less inputs. Thus, we can use less and cheaper and-gates. To summarize: the minimal SOP form realization lead to less and cheaper components and less wiring between these components. All of this leads to a realization with lower cost.

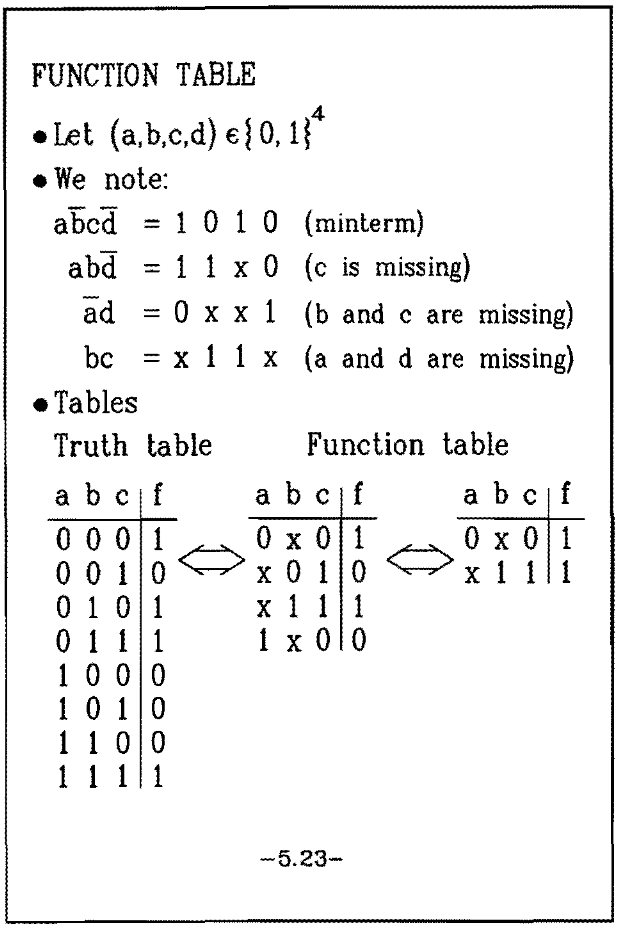
We have previously given a sum-of-minterms realization for the function that is given as the sum of minterms m_0, m_2, m_3 and m_7 (see figure 5.05). In figure 5.21 we shall derive a sum-of-product-terms realization for this function. Here, we first write the function as a sum of minterms. We see that we can delete the variable b from the first two minterms, and the variable a from the last two minterms. This results in a sum-of-products form with two product-terms. A schematic of the realization of this sum-of-products form is shown. It consists of an or-gate with two inputs and two and-gates with two inputs. The previously given realization required more gates with more inputs per gate. There is thus a clear saving.



MINIMIZATION

- Karnaugh diagram
- Quine–McCluskey's method
- Programs
  - espresso algorithm
  - Multiple Output Minimizer (MOM)

-5.22-



FUNCTION TABLE

- Let $(a,b,c,d) \in \{0,1\}^4$
- We note:

$a\bar{b}c\bar{d}$ = 1 0 1 0 (minterm)
$ab\bar{d}$ = 1 1 x 0 (c is missing)
$\bar{a}d$ = 0 x x 1 (b and c are missing)
bc = x 1 1 x (a and d are missing)

- Tables

Truth table

a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

⟺

Function table

a	b	c	f
0	x	0	1
x	0	1	0
x	1	1	1
1	x	0	0

⟺

a	b	c	f
0	x	0	1
x	1	1	1

-5.23-

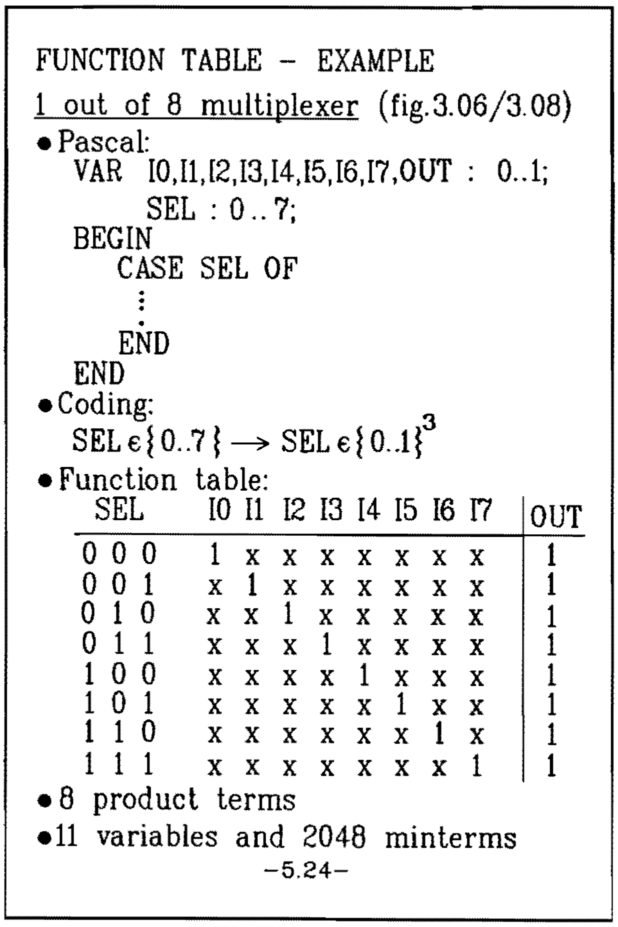
To get the maximum gain from the saving achieved by the sum-of-products form is it necessary that we have means for obtaining the minimal SOP form. Seeking this minimal form is called minimization (see figure 5.22). Among the known methods are Karnaugh diagrams and Quine-McCluskey's method. Both methods are briefly described in practically all books on digital systems design. We shall not do so in this course. Currently, there are a number of good programs available for finding the minimal SOP form. We shall name various implementations of the espresso algorithm. A version of MOM (Multiple Output Minimizer) is suitable for use on a PC. MOM works on a different principle than the espresso algorithm. Notice that non of the programs apply the above mentioned methods of Karnaugh and Quine-McCluskey.

Next to the algebraic method of specifying the sum-of-products form we would also like to have a tabular method, which is comparable with a truth table. Such a table is called a function table (see figure 5.23).

In a function table we do not specify the function's value per minterm but rather per product-term. Also now we describe a product-term by an n-tuple where we need a new symbol, the letter *x*, to denote that the variable corresponding to the specific place in the n-tuple, does not occur in the product-term. Figure 5.23 gives a number of examples. In the first example we see the minterm ab'cd', represented by the quadruple 1010. In the following product-term the variable c is missing. This is denoted by an "x" in the third position in the quadruple. The variables b and c are missing from the product-term a'd, therefore we write 0xx1. The same holds for the product-term bc.

Figure 5.23 shows the truth table of the function f = Σ (0,2,3,7). Next to the truth table, we can also define a function table for the function, where we note the product-terms for which the function equals 1 or 0.

We can also suffice with noting, in the function table, only the product-terms for which the function is equal to 1, omitting the other product-terms for which the function is 0. This form is also shown in figure 5.23. From this we immediately find the function f = a'c' + bc. By doing so we have realized the function of figure 5.21.



FUNCTION TABLE – EXAMPLE

1 out of 8 multiplexer (fig.3.06/3.08)

- Pascal:

```
VAR  I0,I1,I2,I3,I4,I5,I6,I7,OUT : 0..1;
        SEL : 0..7;
BEGIN
   CASE SEL OF
      ⋮
   END
END
```

- Coding:

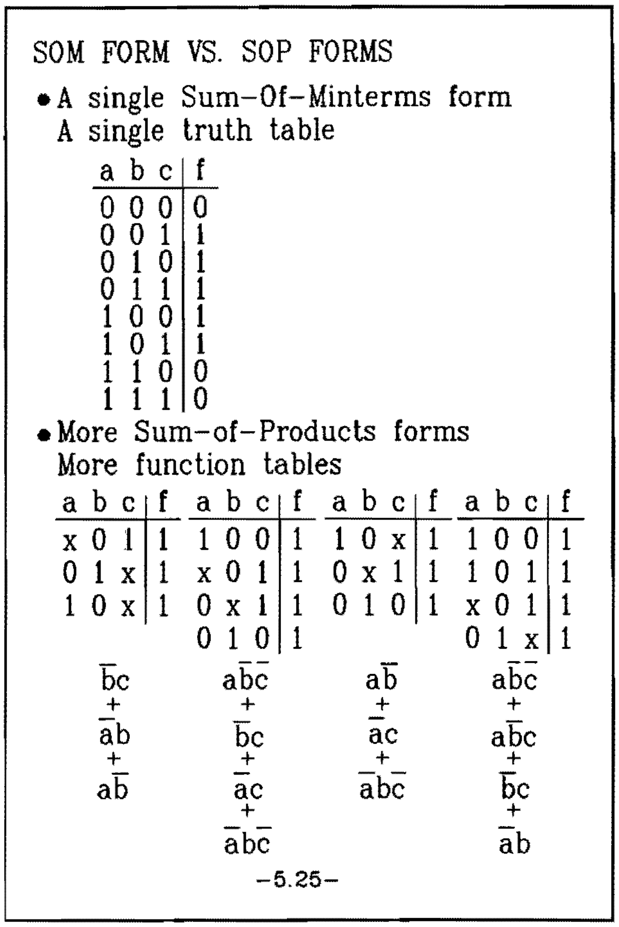
  $SEL \in \{0..7\} \longrightarrow SEL \in \{0..1\}^3$

- Function table:

SEL	I0	I1	I2	I3	I4	I5	I6	I7	OUT
0 0 0	1	x	x	x	x	x	x	x	1
0 0 1	x	1	x	x	x	x	x	x	1
0 1 0	x	x	1	x	x	x	x	x	1
0 1 1	x	x	x	1	x	x	x	x	1
1 0 0	x	x	x	x	1	x	x	x	1
1 0 1	x	x	x	x	x	1	x	x	1
1 1 0	x	x	x	x	x	x	1	x	1
1 1 1	x	x	x	x	x	x	x	1	1

- 8 product terms
- 11 variables and 2048 minterms

–5.24–



SOM FORM VS. SOP FORMS

- A single Sum–Of–Minterms form
  A single truth table

a b c	f
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	1
1 0 0	1
1 0 1	1
1 1 0	0
1 1 1	0

- More Sum–of–Products forms
  More function tables

a b c	f	a b c	f	a b c	f	a b c	f
x 0 1	1	1 0 0	1	1 0 x	1	1 0 0	1
0 1 x	1	x 0 1	1	0 x 1	1	1 0 1	1
1 0 x	1	0 x 1	1	0 1 0	1	x 0 1	1
		0 1 0	1			0 1 x	1

$\bar{b}c + \bar{a}b + a\bar{b}$

$a\bar{b}\bar{c} + \bar{b}c + \bar{a}c + \bar{a}b\bar{c}$

$a\bar{b} + \bar{a}c + \bar{a}b\bar{c}$

$a\bar{b}\bar{c} + a\bar{b}c + \bar{b}c + \bar{a}b$

–5.25–

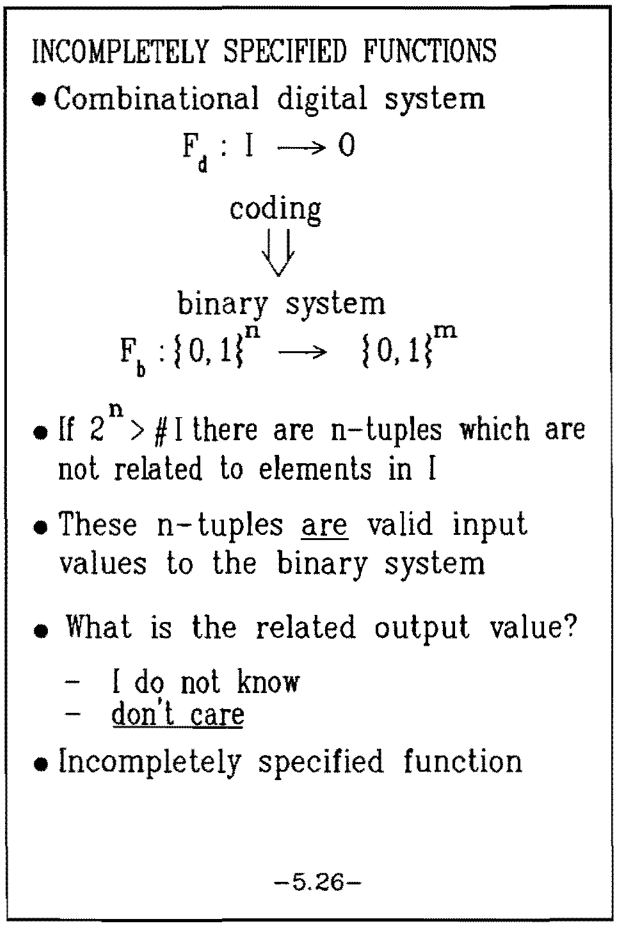
The example in figure 5.24 shows that we can describe functions very compactly using function tables. If we limit ourselves to binary variables for the inputs and outputs of the 1 out of 8 multiplexer of chapter 3, and we code the selection input SEL with a binary triple, then we can simply build up the function table. Indeed, the multiplexer's output OUT is only equal to 1 if the selected input is equal to 1. Thus, if the value of the selection input SEL = 000 then the input i_0 will be switched to the output OUT and OUT will only be 1 if $i_0 = 1$. This is disregarding the value of the rest of the inputs. These values play no role and can be expressed with an x.

Notice that the function table built up in this way only has 8 product-terms, despite the fact that the function has 11 variables and thus has 2048 minterms.

Problem 5.3 (5.3)

Draw a realization with and-gates and or-gates for the function table given in figure 5.24.

In the previous chapter we have noticed that any function has only one specific sum-of-minterms form. Therefore, any function is also specified by a truth table (see figure 5.25). Most functions have more than one sum-of-products form, and therefore more than one function table. In figure 5.25 four different function tables are specified for the same function (the one given by the truth table). The corresponding algebraic sum of products are noted under the function table, so that you can check that it really deals with the same function.



INCOMPLETELY SPECIFIED FUNCTIONS

- Combinational digital system

$$F_d : I \longrightarrow 0$$

coding

$$\Downarrow$$

binary system

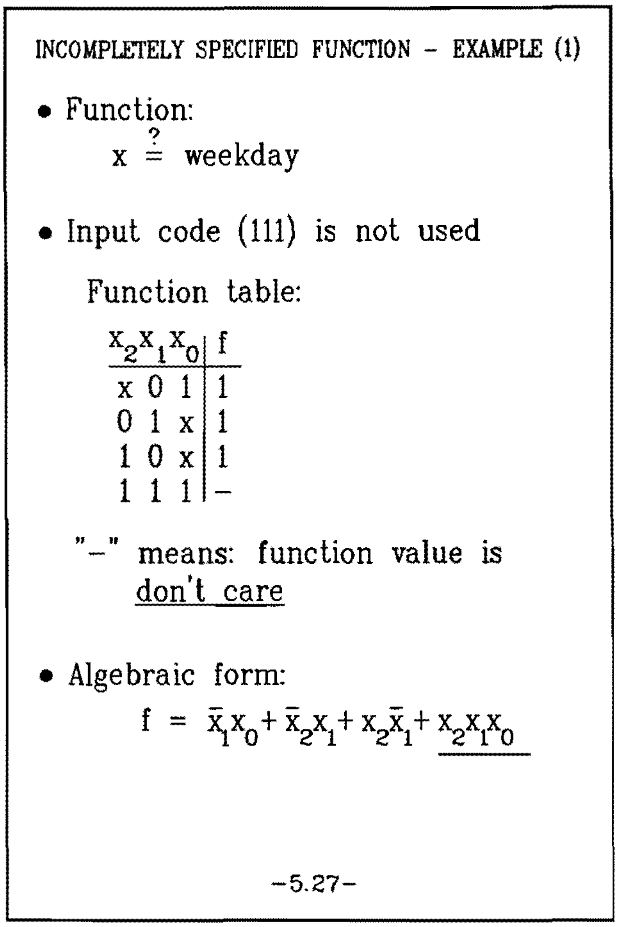
$$F_b : \{0,1\}^n \longrightarrow \{0,1\}^m$$

- If $2^n > \# I$ there are n-tuples which are not related to elements in I
- These n-tuples are valid input values to the binary system
- What is the related output value?
  - I do not know
  - don't care
- Incompletely specified function

-5.26-

In chapter two we have seen that a combinational digital system is described by a mapping from an input domain to an output range. We realize the binary system that is obtained by coding the digital system (see figure 5.26). Using this binary system, a binary n-tuple at the input is mapped to a binary m-tuple at the output. With n input bits we can make 2^n different codes. This number must be at least equal to the number of elements in our inputs set I. If the number of codes is larger than the number of elements in I, then there are n-tuples that are not used to code elements of I.

These n-tuples are valid input values for the binary system. They are not related to elements of I. What is then the corresponding output value of the binary system? There is no answer to this question: in this case the desired system behaviour does not specify a value for the output m-tuple. Accordingly we must say that we do not know what the output value must be. In other words it does not really matter; the value is a *don't care* value. Functions of which value is not defined for certain input n-tuples are called *incompletely specified functions.*



INCOMPLETELY SPECIFIED FUNCTION – EXAMPLE (1)

- Function:
  $x \overset{?}{=}$ weekday

- Input code (111) is not used

  Function table:

$x_2 x_1 x_0$	f
x 0 1	1
0 1 x	1
1 0 x	1
1 1 1	–

"–" means: function value is <u>don't care</u>

- Algebraic form:

$$f = \bar{x}_1 x_0 + \bar{x}_2 x_1 + x_2 \bar{x}_1 + \underline{x_2 x_1 x_0}$$

–5.27–

We have coded the 7 days of the week with a 3-bit code in the "weekday" function. Consequently there is an unused code from the 8 possible code words (see figure 5.27). This is code-word 111. The value of the function for this code is "don't care". Previously, we have shown this in the truth table by writing 0/1 in the place for the function value. Form now on we shall denote these don't cares by a horizontal dash in the function table.

Figure 5.27 shows the modified function table for the "weekday" function. We point out with emphasis that this don't care hyphen is not a possible output value of the function, but it only shows that we did not specify what the output value must be for these input combinations. When the function is realized *we shall have to make a choice.* We shall come back to this point later. In an algebraic expression of the sum of products we can also show that the function's output is don't care in the case of a specific product-term. We do this by underlining the corresponding product-term. Figure 5.27 shows an example.

Problem 5.4 (5.4)
Figure 2.28 gives a gate realization of the "weekday" function. Verify that this realization corresponds to the function table of figure 5.27. What do you notice in relation to the don't care value?

INCOMPLETELY SPECIFIED FUNCTION – EXAMPLE (2)

- Let $i \in \{0..9\}$ and $f \in \{0,1\}$
  f=1 if i is an even number
- Use binary code
  0 ⟶ 0000
  1 ⟶ 0001
  ⋮
  9 ⟶ 1001
- Unused codes

1010	1101
1011	1110
1100	1111

- Function table

d c b a	f	
0 x x 0	1	(0,2,4,6)
1 0 0 0	1	(8)
1 0 1 x	–	
1 1 x x	–	

$$f = \bar{d}\bar{a} + d\bar{c}\bar{b}\bar{a} + \underline{d\bar{c}b} + \underline{dc}$$

–5.28–

In figure 5.28 we see the following example: A digital system which has an input with a value domain of the integer numbers from 0 to 9, and an output with a value range 0 or 1. The function between the input and output is given as

"f = 1 when the value of i is even"

If we use a binary code of 4 bits to code the possible input values, then there will remain 6 unused codes, corresponding to the hexadecimal digits A to F. Figure 5.28 shows the function table for f. We shall not discuss the derivation of this function table. The algebraic sum-of-product terms are also given, with the product-terms leading to a don't care value underlined. If we make the function value for these product-terms 0, we shall get a function with 2 product-terms. Accordingly we need a total of 3 gates to realize this function.

INCOMPLETELY SPECIFIED FUNCTION – MINIMIZATION

- With minimization a completely specified function is produced with
  - a minimum number of product terms
  - each product term having a minimum number of variables
- During minimization don't care terms are made 0 or 1
- Example
  - f=1 if i is an even number
  - function table after MOM

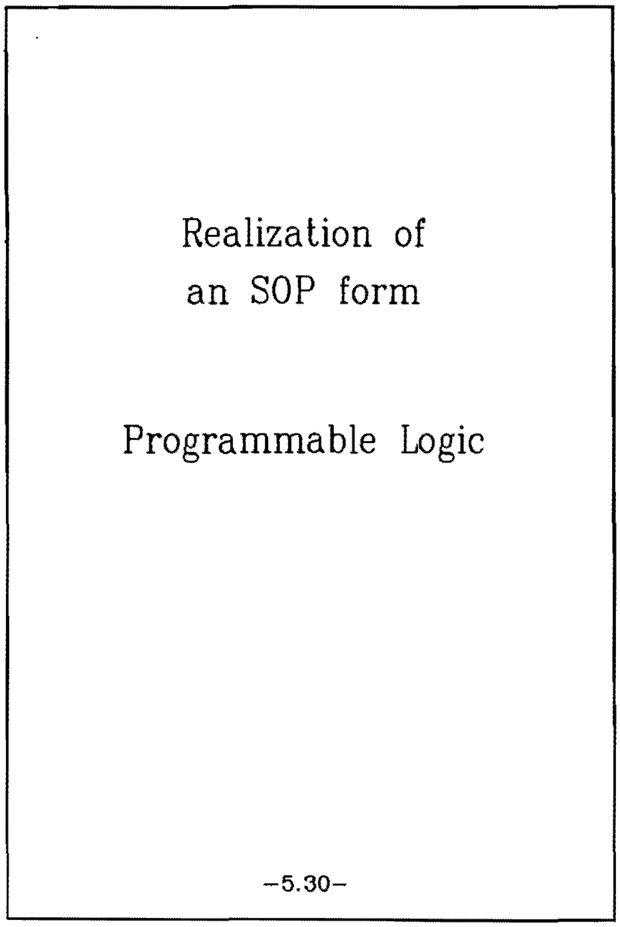
d c b a	f
x x x 0	1

  - minimized function: $f = \bar{a}$
- In the minimized version f = 1 also for the quadruples

  1010, 1100, 1110

–5.29–

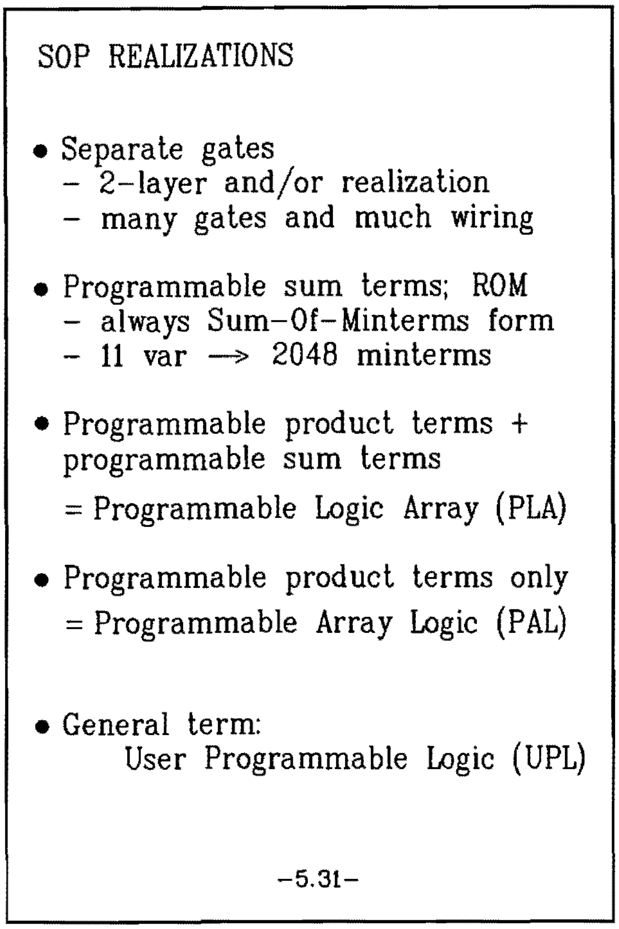
A function with don't care terms gives us an extra degree of freedom when realizing it. For each don't care term we have the choice of realizing the function value as a 0 or a 1. Good minimization methods and programs (figure 5.29) make use of these extra degrees of freedom to find the smallest possible number of product-terms. Minimization converts an incompletely specified function into a completely specified one, with all product-terms containing the smallest possible number of variables. Thus, an automatic optimal choice is made for the don't care terms. Figure 5.29 shows the resulting function table for the function "f = 1 if i is even" after minimization by the MOM program. During the minimization it turns out that such an optimal solution is obtainable if the function is equal to 1 for the inputs 1010, 1100 and 1110. In that case the function can be realized with a single inverter. We see that don't care terms can play an important role in finding optimal realizations for switching functions. Therefore, we have to be alert for possible don't care situations.



Realization of
an SOP form

Programmable Logic

–5.30–



SOP REALIZATIONS

- Separate gates
  - 2-layer and/or realization
  - many gates and much wiring
- Programmable sum terms; ROM
  - always Sum-Of-Minterms form
  - 11 var —> 2048 minterms
- Programmable product terms + programmable sum terms

  = Programmable Logic Array (PLA)
- Programmable product terms only

  = Programmable Array Logic (PAL)
- General term:

  User Programmable Logic (UPL)

–5.31–

5.3 Realization of an SOP form; programmable logic

We have discussed two realization methods for the sum-of-products form. Figure 5.31 states them once more.

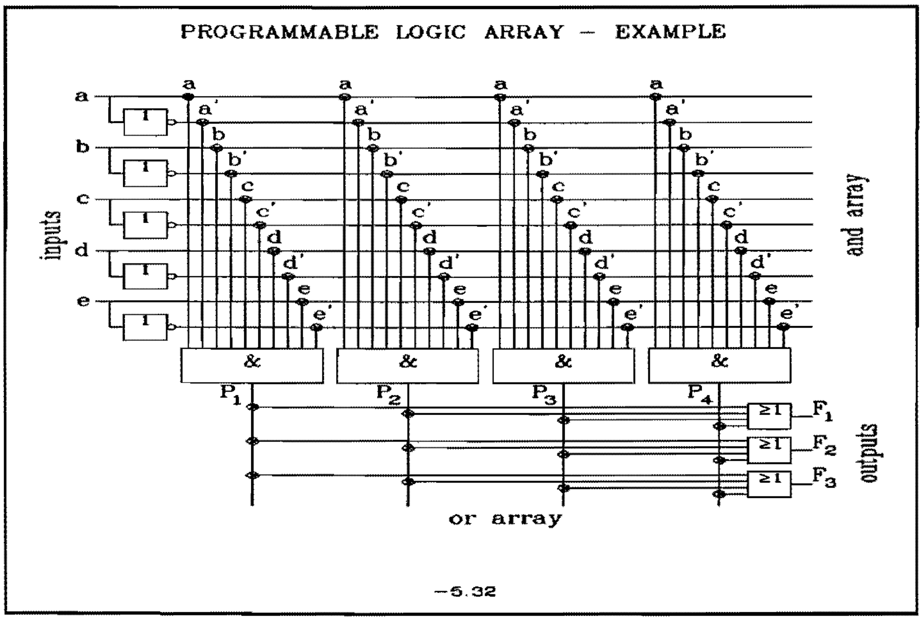
First we can realize a sum of products by means of a two-layer and-or realization. Therefore a number of interconnected discrete building blocks are necessary. Thus, lots of components with lots of wiring.

We can also chose for a realization with a ROM. Then we must realize the function as a sum of minterms. This can sometimes be infeasible. Earlier we have composed the function table for the binary 1-out-of-8 multiplexer. We stated that the table contained only 8 product-terms, while the function had 11 variables.

In a ROM implementation the function value must be specified for 2048 minterms. This is a consequence of the fact that we can only program sum-terms in a ROM. On the other hand the product-terms to be used are fixed when minterms are realized in the address decoders.

We get a more flexible building block by also making the composition of product-terms programmable. Such a building block is called a PLA, a *Programmable Logic* Array, being very complex and expensive.

We get a less complex building block which is also very useful for the realization of a sum-of-products form, by making the sum terms fixed and only the product-terms programmable. In this case we are considering a PAL, *Programmable Array Logic.* In general terms programmable building blocks are known as *User Programmable Logic,* UPL.



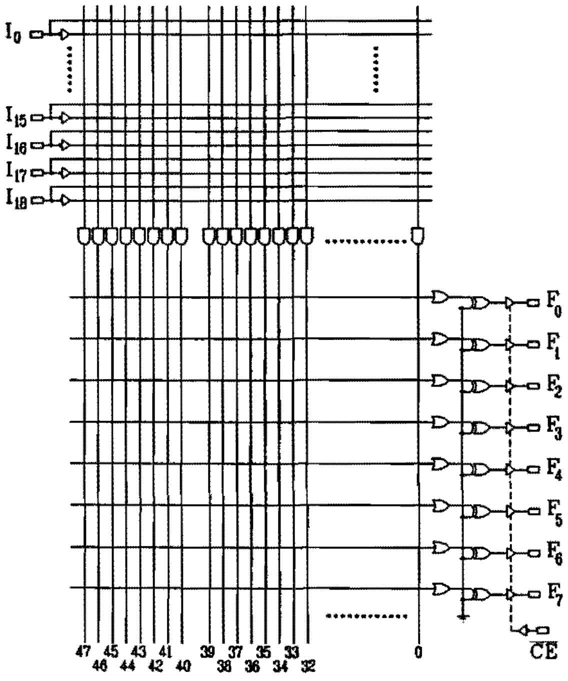
PROGRAMMABLE LOGIC ARRAY – EXAMPLE

–5.32

In this paragraph we shall take a brief look at the construction and the use of the PLA and the PAL. In principle the PLA consists of two large programmable matrices (see figure 5.32), the and-matrix and the or-matrix.

In the and-matrix we can denote which variables (or inverted variables) should be selected to compose a product-term, by using a number of programmable connections. In the or-matrix we combine these different product-terms into a sum-of-product terms.

FPLA 82S100/82S101 LOGIC DIAGRAM



-5.33-

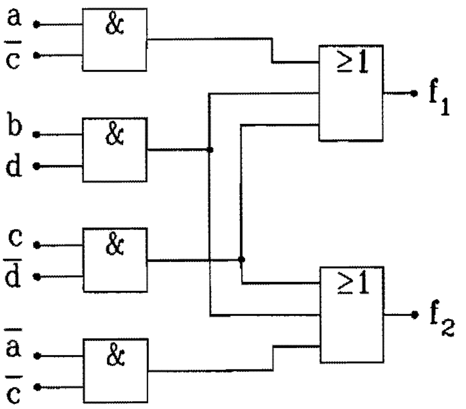
REUSE OF PRODUCT TERMS

- Functions

$f_1 = a\bar{c} + bd + c\bar{d}$

$f_2 = bd + c\bar{d} + \overline{ac}$

- Realization



- Multiple Output Minimization (MOM)

-5.34-

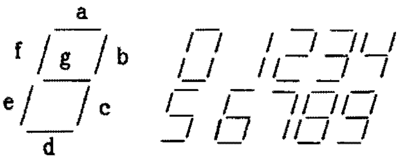
Again we apply a simplification when drawing schematics of this type. We only draw one line to the and-gates and the or-gates. We get a schematic such as the one shown in figure 5.33. As we have done in the ROM, we show here that a connection must be made between an input and an and-gate, or a product-term and an or-gate with a dot on a crossing of wires.

In the schematic of figure 5.33 we also see that in a PLA it is possible to use a specific product-term for several output functions in the same time. We are then considering *product-term sharing*. This reuse of product-terms enables us to use the building block more efficiently.

Figure 5.34 gives an example of this. The two functions f_1 and f_2 are shown, each defined as a sum of three product-terms. We see that the product-terms bd and cd' are used in both functions. We can thus realize four product-terms instead of six. But we must then connect the outputs of two and-gates with both or-gates of both the function (f_1 and f_2). In general this is not a problem. To be able to make optimal use of the reusability of product-terms it is necessary to actively seek sum-of-products forms for several functions where the same product-terms are used as much as possible.

REUSE OF PRODUCT TERMS - EXAMPLE

- BCD code to 7 segment code converter



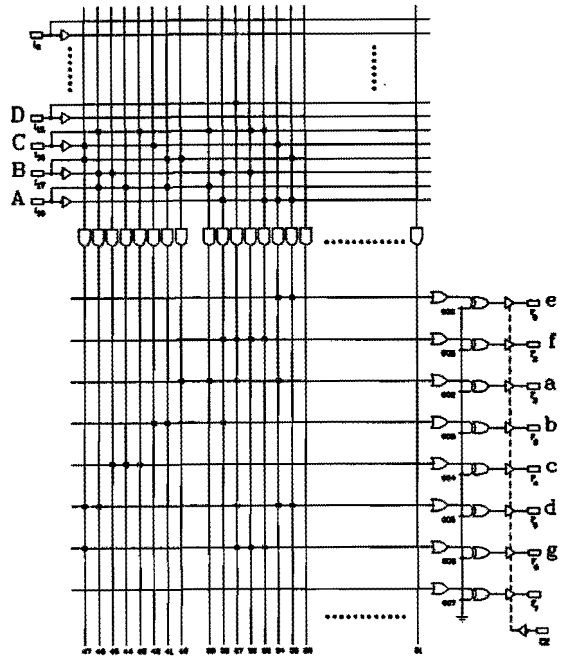
- Truth table

D	C	B	A	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	-	-	-	-	-	-	-
1	0	1	1	-	-	-	-	-	-	-
1	1	0	0	-	-	-	-	-	-	-
1	1	0	1	-	-	-	-	-	-	-
1	1	1	0	-	-	-	-	-	-	-
1	1	1	1	-	-	-	-	-	-	-

-5.35-

SINGLE OUTPUT MINIMIZATION - EXAMPLE WITH PLA

- BCD code to 7 segment code converter



-5.36-

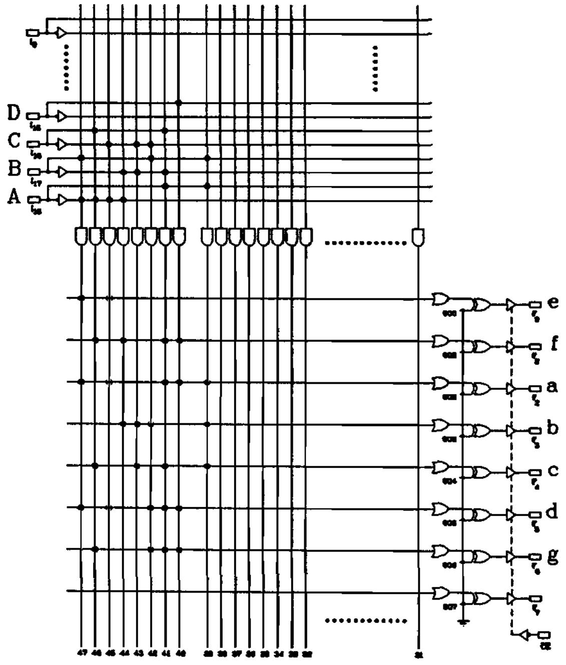
Some minimization programs can do this; we refer to *Multiple Output Minimization* (MOM). As an example of the reusability of product-terms we define in figure 5.35 a BCD-code to 7-segment-code translator. A 7-segment-code is used in a 7-segment display, being able to display the digits 0 to 9 with the help of 7 segments. Figure 5.35 shows how the 7 segments could be used. Now, the truth table can be simply written.

Notice that for the input codes 1010 to 1111, which do not belong to the BCD code, the outputs of the code translator are don't cares.

Figure 5.36 gives a PLA implementation for this code translator, where single output minimization is applied.

MULTIPLE OUTPUT MINIMIZATION -
EXAMPLE WITH PLA

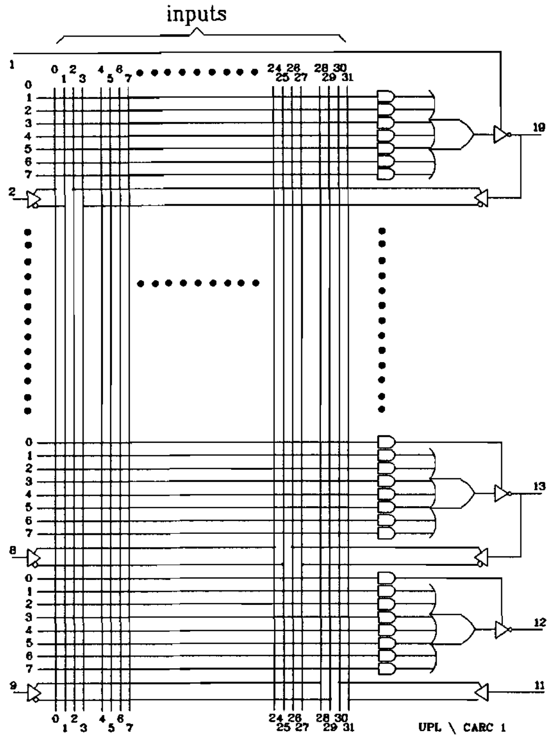
- BCD code to 7 segment code converter



-5.37-

PAL 16L8 LOGICAL DIAGRAM

- Only and-array is programmable



-5.38-

Figure 5.37 gives the code translator realized as a PLA, where multiple output minimization is applied. Notice that now clearly less product-terms are necessary. In borderline cases this means that we only need 1 instead of 2 building blocks for the realization.

We shall close this paragraph by showing another structure in figure 5.38: a PAL. In the PAL only the product-terms are programmable, and it is determined in advance which product-terms are going to belong to which sums, and hence to which outputs. The number of product-terms per output is therefore fixed. The form of the product-terms is programmable. Generally, the reuse of product-terms is not possible in a PAL. Therefore, multiple output minimization is pointless, stronger still it is not advisable. Single output minimization gives better results in these cases.

Realization with

NAND and

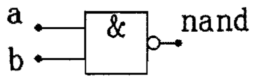
NOR gates

-5.39-

NAND GATE

- Truth table and gate symbol

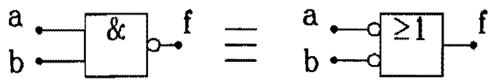
a	b	nand
0	0	1
0	1	1
1	0	1
1	1	0



$$f = \overline{a \cdot b} = (a \cdot b)'$$

- De Morgan's law

$$f = \overline{a \cdot b} = \overline{a} + \overline{b}$$



-5.40-

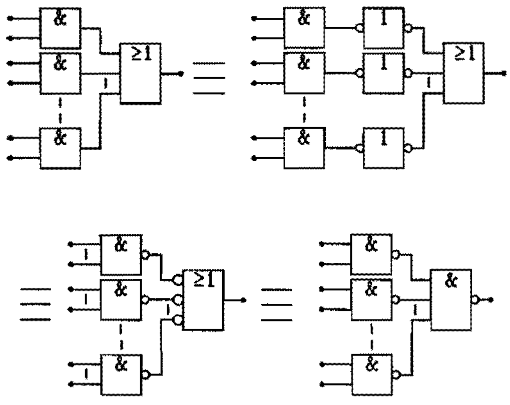
5.4 Realization with NAND and NOR gates

We have seen that any function can be described as a sum of products. This sum-of-products form can be realized by making use of and-gates, or-gates and inverters. Therefore, there are three necessary building blocks. We shall now show that these functions can also be realized using only one type of building block. This building block is the *nand-gate.*

Figure 5.40 shows the characteristics of the nand-gate given in the form of a truth table and an algebraic expression. Notice that the nand-gate can be obtained from an and-gate by inverting its output. We are thus considering the not-and-gate. The standardized symbol corresponds to this statement. Another representation can be obtained by applying the rules of De Morgan to the algebraic expression. Indeed, De Morgan states that the inverse of a · b equals a' + b'. This is an or-relation between the complements of two variables. This equivalence is shown at the bottom of figure 5.40.

SOP REALIZATION WITH NAND GATES

- Any SOP realization has an equivalent with only NAND gates
- Why:

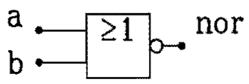


-5.41-

NOR GATE

- Truth table and gate symbol

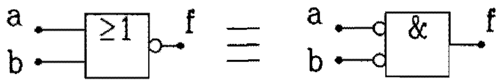
a b	nor
0 0	1
0 1	0
1 0	0
1 1	0



$$f=\overline{a+b}=(a+b)'$$

- De Morgan's law

$$f=\overline{a+b}=\bar{a}\bullet\bar{b}$$



-5.42-

By making use of this equivalence we can now state the following (see figure 5.41):

Any SOP realization has an equivalent realization which only uses nand-gates.

This assertion is simple to explain if we consider that the cascade connection of two inverters has no effect on the final function. These inverters might also be placed at the outputs of the and-gates and at the inputs of the or-gate of a SOP realization. We have just seen that an or-gate with inverters at its inputs is equivalent to a nand-gate. Herewith, the assertion proof is almost done.

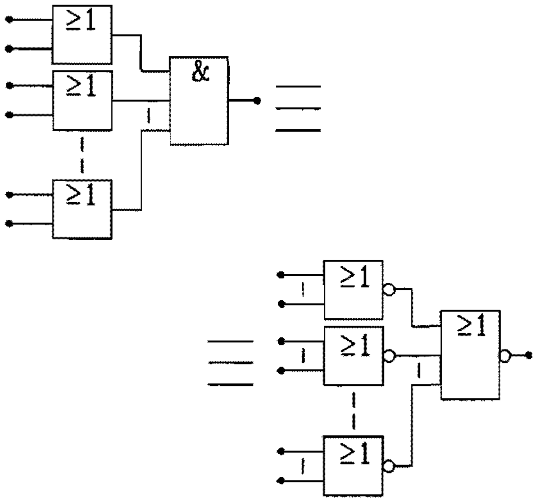
In a number of cases we still need extra inverters. Here we can also use a nand-gate. Indeed, if we connect both inputs of a 2 input nand-gate to each other the gate functions as an inverter. Check this for yourself.

Next to the nand-gate we also know the nor-gate, which is obtainable form the or-gate by inverting its output: the not-or-gate. Figure 5.42 shows the characteristics and the symbol of this gate. We also see that, using De Morgan's rules, we can view this gate as an and-gate with inverted inputs.

In this chapter we have only considered the sum of minterms and the sum-of-product terms. We have, to some extent, overlooked the existence of maxterms and general sum terms. Actually, according to the duality principle, for any function we can construct a product-of-maxterms form, or a more general product-of-sums form. This can be realized using a 2-layer or-and realization, where the output results from the and-gate and the inputs of the and-gate are connected to an equivalent number of or-gates.

POS REALIZATION WITH NOR GATES

- Any SOP realization has an equivalent realization with only NOR gates
- Why:

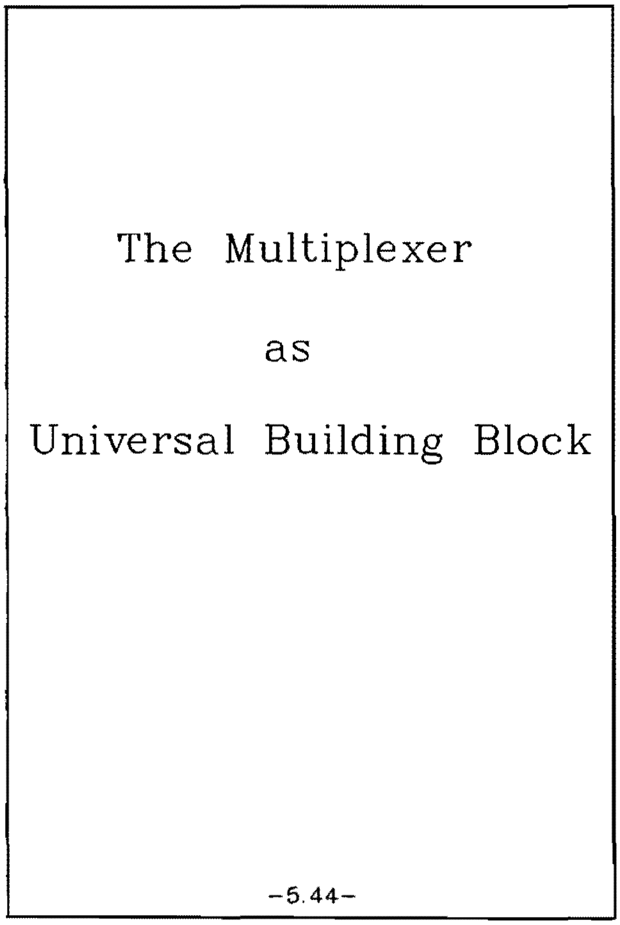


-5.43-

In relation to the nor-gate we can now make the following statement about the product of sums form (figure 5.43):

> *Any POS realization has an equivalent realization which only uses nor-gates.*

The left top part of figure 5.43 shows a POS realization. As we have done with the SOP realization, we can transform this schematic into the schematic at the right bottom of figure 5.43, where we still only use nor-gates. We see that an arbitrary function is realizable using only nand-gate or only nor-gates. This is important, because in practice it is easier and cheaper to make a nand-gate or a nor-gate than an and-gate or an or-gate with electronic devices. Frequently an and-gate is realized by taking a nand-gate and putting an inverter behind it. Realizations with nand-gates or nor-gates are in general cheaper and faster.

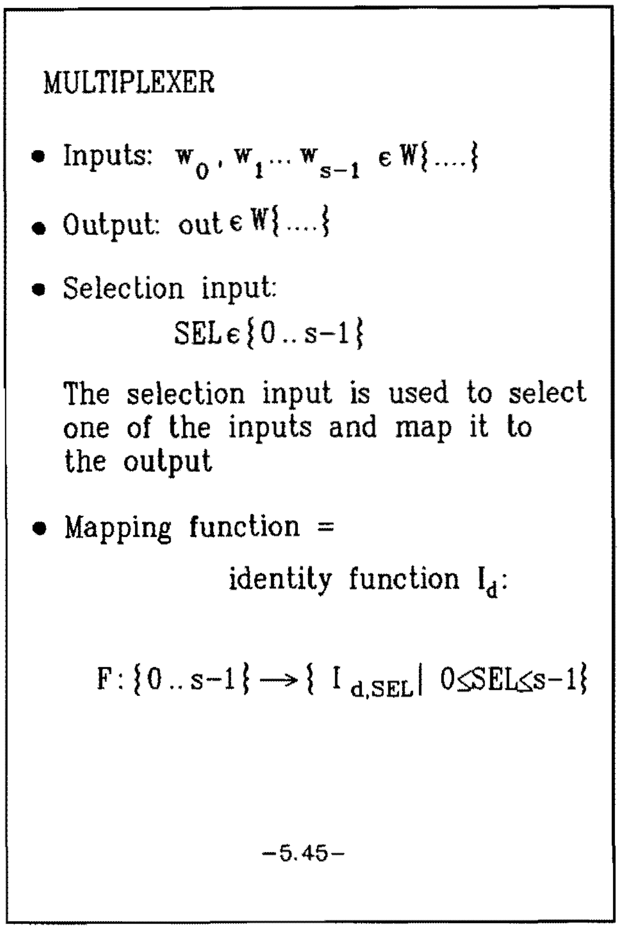


The Multiplexer

as

Universal Building Block

-5.44-



MULTIPLEXER

- Inputs: $w_0, w_1 \ldots w_{s-1} \in W\{\ldots.\}$
- Output: out $\in W\{\ldots.\}$
- Selection input:

  $SEL \in \{0 .. s-1\}$

  The selection input is used to select one of the inputs and map it to the output
- Mapping function =

  identity function I_d:

$$F: \{0 .. s-1\} \longrightarrow \{ I_{d,SEL} \mid 0 \leq SEL \leq s-1\}$$

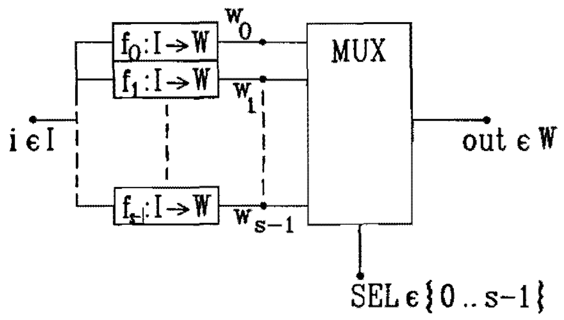
-5.45-

5.5 The multiplexer as a universal building block

In chapter 3 (figure 3.06) we got acquainted with the multiplexer. Under control of a selection input SEL, one of the inputs is connected to the output.

We have stated that the control input, SEL, determines the function of the multiplexer and indirectly determines the value of the output (see figure 5.45). With the control input, SEL, we select the mapping of one the inputs to the output.
We can show this by using the identity function I_d. The value of this function is equal to its argument.

MUX INPUTS = FUNCTION RESULTS



- We select a function $f_{SEL}:I\to W$
- Total function $F_{tot}:\{0..s-1\}\to(I\to W)$ leading to $F_{tot}(SEL) = f_{SEL}$
- Alternative representation of total function: $F_{tot}: I * \{0..s-1\}\to W$
- Can each of the functions $f_{SEL}:I\to W$ be realized in a simpler way?

-5.46-

It becomes interesting if we allow the input values of the multiplexer to be results of functions. We have expressed this schematically in figure 5.46. We now have an input domain, I, and a collection of s functions f_0 to f_{s-1}.

All of these functions map their input set to the value set W of the multiplexer inputs. We select one of the functions f_j with the selection input SEL. The total system can be described by mapping the value set of the selection input, SEL, onto a set of functions that map the input domain I onto the output range W, in a way that we select one of the functions f_{SEL} for each value of the section input SEL. An equivalent representation is the mapping of the product of the input domain I and the value range of the selection input to the output range W. We have turned the realization of a function into the realization of another s functions. We hope now that the realization of each of these functions is simpler that the realization of the original function.

BINARY 1 OUT OF 2^m MULTIPLEXER (1)

- Domains: out $\in W=\{0,1\}$
  SEL $\in\{0,1\}^m$
- Assume $I=\{0,1\}^n$
- Accordingly
  $F_{tot}:\{0,1\}^m\to(\{0,1\}^n\to\{0,1\})$
  $F_{tot}:\{0,1\}^n * \{0,1\}^m\to\{0,1\}$
- Hence
  $F_{tot}:\{0,1\}^{n+m}\to\{0,1\}$
- Assertion
  A combinational binary system with
  $F:\{0,1\}^{n+m}\to\{0,1\}, n\geq 0, m>0$
  can be realized with
  1 out of 2^m multiplexer

-5.47-

Finally, considering our switching functions, we turn our attention to the binary multiplexer. Figure 5.47 shows that for this multiplexer is it valid that the output has two values, i.e. it is binary. Furthermore, the 1 out of 2^m multiplexer has m binary selection inputs. The value of SEL is now coded in an n-tuple. If we form the value range of the inputs, I, from a set of n-tuples, then we can realize any arbitrary combinational switching function of n + m inputs with a binary multiplexer. The precise assertion is shown in figure 5.47.

BINARY 1 OUT OF 2^m MULTIPLEXER (2)

- Accordingly

$$F: \{0,1\}^n \rightarrow \{0,1\}$$

can be realized with

$$F: \{0,1\}^m \rightarrow (\{0,1\}^{n-m} \rightarrow \{0,1\})$$

alternatively:

$$F: \{0,1\}^m \rightarrow (F')$$

$$F': \{0,1\}^{n-m} \rightarrow \{0,1\}$$

(2^m different functions F')

We distinguish

- n-m >0

2^mfunctions of n-m variables

- n-m =0

2^m functions $F': \{\varepsilon\} \longrightarrow \{0,1\}$

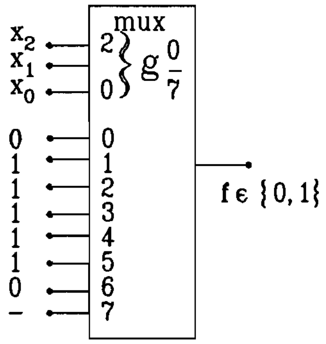
These are constant values!

-5.48-

REALIZATION WITH MULTIPLEXER-EXAMPLE (1)

- Function: is x a weekday?
- Truth table

$x_2x_1x_0$	f	$x_2x_1x_0$	f
0 0 0	0	1 0 0	1
0 0 1	1	1 0 1	1
0 1 0	1	1 1 0	0
0 1 1	1	1 1 1	-



$$f: \{0,1\}^3 \longrightarrow (\{\varepsilon\} \longrightarrow \{0,1\})$$

-5.49-

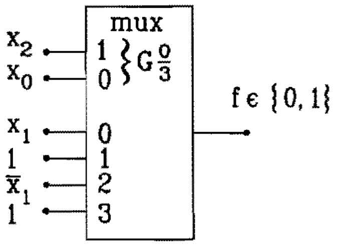
In figure 5.48 we have formulated this in another way. Let F be a mapping of an n-tuple to a binary value. This function can then be realized with the help of a 1 out of 2^m multiplexer selecting a function from a set of functions mapping an (n-m)-tuple to {0,1}. This set contains a maximum of 2^m different functions, F'. Now we can distinguish between two cases. First n-m may be larger than 0. Then we have 2^m functions of n-m variables. If n-m equals 0 then we have 2^m functions that form a mapping from an empty set to {0,1}. These are constant values.

Figure 5.49 shows an example. There the function "is x a weekday" is realized using a 1 out of 8 multiplexer. Notice that the three variables x_2 to x_0 are now necessary for the control of the selection input of the multiplexer. The functions of the other inputs are thus constants. We now see that we can place the function values of the truth table on these inputs.

REALIZATION WITH MULTIPLEXER-EXAMPLE (2)

1 out of 4 multiplexer with selector x_2x_0

x_2x_0	x_1	f	
0 0	0	0	$f'_0 = x_1$
	1	1	
0 1	0	1	$f'_1 = 1$
	1	1	
1 0	0	1	$f'_3 = \bar{x}_1$
	1	0	
1 1	0	1	$f'_4 = 1$
	1	-	



$f : \{0,1\}^2 \rightarrow (\{0,1\} \rightarrow \{0,1\})$

-5.50-

In figure 5.50 we have realized the same function using a 1 out of 4 multiplexer. We now need two variables to select a function of a third variable. In our example we chose x_2 and x_0 as selection variables. By writing the truth table somewhat differently we can directly read from it which function of x_1 must be set on the other inputs of the multiplexer.

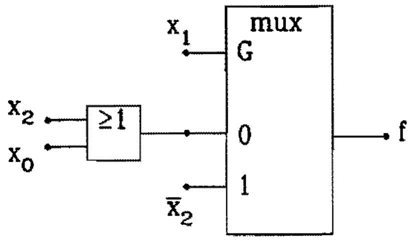
REALIZATION WITH MULTIPLEXER-EXAMPLE (3)

- 1 out of 2 multiplexer with selector x_1

x_1	x_2 x_0	f
0	0 0	0
	0 1	1
	1 0	1
	1 1	1
1	0 0	1
	0 1	1
	1 0	0
	1 1	-

$$f'_0 = \begin{cases} 1 & \text{if } (x_2x_0) \in \{01,11,10\} \\ 0 & \text{otherwise} \end{cases}$$

$$f'_1 = \begin{cases} 1 & \text{if } (x_2x_0) \in \{00,01\} \\ - & \text{if } (x_2x_0) = (1,1) \\ 0 & \text{otherwise} \end{cases}$$



$f : \{0,1\} \longrightarrow (\{0,1\}^2 \longrightarrow \{0,1\})$

-5.51-

In figure 5.51 we have finally once more realized the same function, but now with a 1 out of 2 multiplexer where we have used the variable x_1 as a selection variable. By rearranging the truth table we can once more determine the function that we have to set on both of the other inputs of the multiplexer. Notice that the function f_0' is 1 if either x_2 or x_0 are equal to 1. The function f_1' equals 1 if x_2 equals 0 (after making the correct choice of the don't care value). Check this for yourself. This produces the realization shown in figure 5.51.

Problem 5.5 (5.5)

For the realization of the function "is x a weekday" using a 1 out of 2 multiplexer, we can also choose x_2 or x_0 as selection variables instead of x_1. Determine for both of these alternative realizations what the other multiplexer input functions must be.

The

exclusive−OR

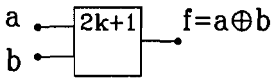
function

−5.52−

THE EXCLUSIVE−OR FUNCTION (1)

- 2 variables

a	b	f
0	0	0
0	1	1
1	0	1
1	1	0

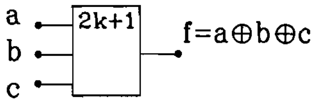


$$f=\begin{cases}1 & \text{if } a \neq b\\ 0 & \text{otherwise}\end{cases}$$

$$f=\bar{a}b+a\bar{b}$$

- 3 variables

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



$$f=\begin{cases}1 & \text{if 2k+1 inputs are 1}\\ 0 & \text{otherwise}\end{cases}$$

−5.53−

5.6 The exclusive OR function

Next to the and, or, nand and nor-gates there is a fifth gate that cannot be ignored in a chapter about the realization of switching functions. This is the exclusive-or gate. This gate realizes the exclusive-or function. Figure 5.53 shows the truth table of the exclusive-or function of two variables and its corresponding symbol. We say also

f = a exclusive-or b

Or:

$$f = a \oplus b$$

We see that the function equals 1 if a is not equal to b, or:

f = a'b + ab'

Otherwise the function equals 0. We can also define an exclusive-or function for more that two variables. In figure 5.53 we have the truth table for the exclusive-or function for three variables. We can now say that the exclusive-or function is equal to 1 if an odd number of inputs is equal to 1. This odd number is expressed with 2k+1. This explains the symbol of the exclusive-or function.

THE EXCLUSIVE-OR FUNCTION (2)

- Assertion:

  An exclusive -OR function of n variables

  $$x_1 \oplus x_2 \oplus \dots . \oplus x_n$$

  has a minimal SOP form with

  2^{n-1} product terms

- This is the SOM form

-5.54-

THE EXCLUSIVE-OR FUNCTION (3)

Realization as a binary tree

- Assume $f = x_1 \oplus x_2 \oplus \dots . \oplus x_{n-1} \oplus x_n \oplus \dots . \oplus x_{2n}$

  Let $f_l = x_1 \oplus x_2 \oplus \dots . \oplus x_n$

  and $f_h = x_{n+1} \oplus \dots . \oplus x_{2n}$
- The following holds:
  - if an odd number of terms $x_1 .. x_{2n}$ have the value 1
  - then f=1
- Hence f=1 if *either*

  an <u>odd</u> number $x_1 .. x_n = 1$

  and

  an <u>even</u> number $x_{n+1} .. x_{2n} = 1$

  or

  an <u>even</u> number $x_1 .. x_n = 1$

  and

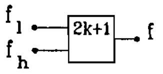
  an <u>odd</u> number $x_{n+1} .. x_{2n} = 1$
- Accordingly f=1 if either

  $f_l = 1$ and $f_h = 0$

  or

  $f_l = 0$ and $f_h = 1$
- The function f can be rewritten:

  $f = f_l \cdot \bar{f}_h + \bar{f}_l \cdot f_h = f_l \oplus f_h$



-5.55-

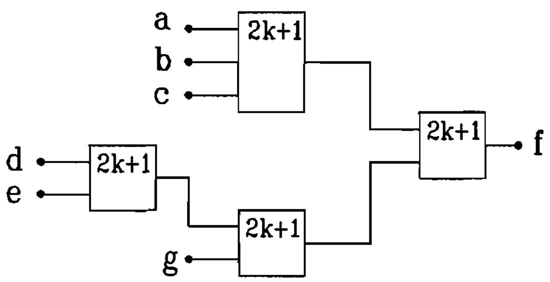
The exclusive-or function has a drawback which is summarized in the statement of figure 5.54. An exclusive-or function of n variables has a minimum sum-of-products form with 2^{n-1} product-terms. All these product-terms are minterms. These minterms are formed by all binary n-tuples with an odd number of elements that are equal to 1. These minterms can not be combined into a smaller number of product-terms. Consequently, an exclusive-or function of a larger number of variables can not be optimally realized as a 2-layer and-or network. Indeed, we need a separate and-gate for each minterm.

Luckily, it is possible to realize exclusive-or functions of many variables as a binary tree. This is shown in figure 5.55. There our starting point is an exclusive-or function of 2n variables. This is split into two exclusive-or functions f_l and f_h, each having n variables. See figure 5.55. We know that the function f = 1 if an odd number of variables x_1 to x_{2n} = 1. This condition can be translated into equivalent conditions for the variables x_1 to x_n and the variables x_{n+1} to x_{2n} (see figure 5.55). We can then conclude that the function f equals 1 if the function f_l = 1 and f_h = 0 or f_l = 0 and f_h = 1. If we compare this with our exclusive-or function, then we see that f equals $f_l \oplus f_h$. Thus, the first layer of our binary tree is formed by an exclusive-or gate.

THE EXCLUSIVE-OR FUNCTION (4)

Example 1

$$f = a \oplus b \oplus c \oplus d \oplus e \oplus g$$



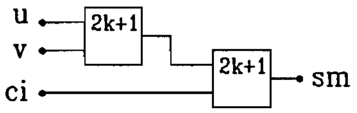
-5.56-

THE EXCLUSIVE-OR FUNCTION (5)

Example 2 - ADD2 module

ci	u	v	sm	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$sm = ci \oplus u \oplus v$$



$$\begin{aligned} co &= \Sigma(3,5,6,7) \\ &= \overline{ci} \cdot u \cdot v + ci \cdot u \cdot v + ci \cdot \overline{u} \cdot v + ci \cdot u \cdot \overline{v} \\ &= uv + ci(u \oplus v) \end{aligned}$$

-5.57-

The resulting f_l and f_h can now be further divided in the same way into smaller exclusive-or functions. By using 2 and 3-input exclusive-or gates we can realize exclusive-or functions of more variables, such as the shown in figure 5.56.

We shall now discuss two practical examples where the exclusive-or function is quite suitable. First we have once more the truth table of our ADD2 module in figure 5.57.

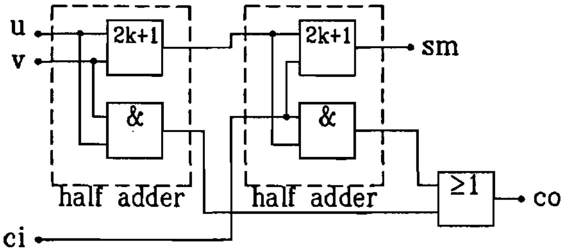
Comparing the column of the sm function with the truth table of an exclusive-or function of three variables, we see that sm is given by the exclusive-or of ci, u and v:

$$sm = ci \oplus u \oplus v$$

Thus, the sum output (sm) can be realized with a 3-input exclusive-or gate, or two 2-input exclusive-or gates as shown in figure 5.57. We can also see that when realizing the carry-out output, co, we can also make use of the exclusive-or between the variables u and v. Because we have already used this exclusive-or function to determine the sum, we can do with an or-gate and two and-gates for the realization of the co function.

THE EXCLUSIVE-OR FUNCTION (6)

Example 3 - full adder



half adder:

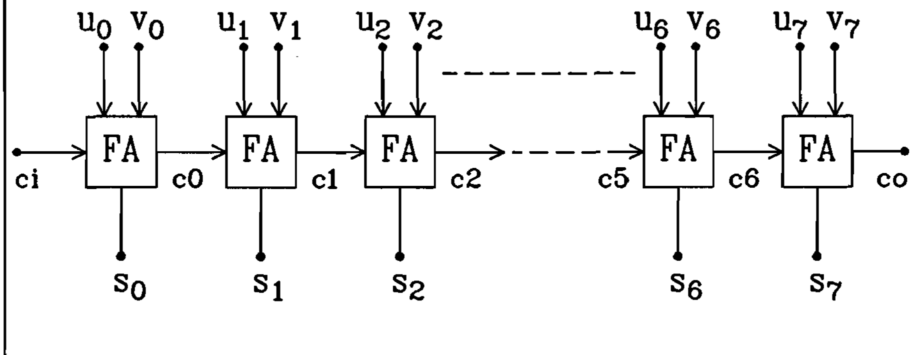
u v	s c
0 0	0 0
0 1	1 0
1 0	1 0
1 1	0 1

-5.58-

Figure 5.58 shows the whole realization of the ADD2 module. We recognize the two exclusive-or gates for the determination of the sum, sm, and the two and-gates and an or-gate for the determination of the co function. Such a circuit is called a "full adder".

The circuit surrounded by a dotted line consisting of an exclusive-or gate and an and-gate is called a "half-adder". Figure 5.58 shows the truth table of this half-adder. We see that this half-adder determines the binary sum of two bits, without adding a carry bit from a previous addition.

8-BIT ADDER AS AN 8-ITERATIVE CIRCUIT



-5.59-

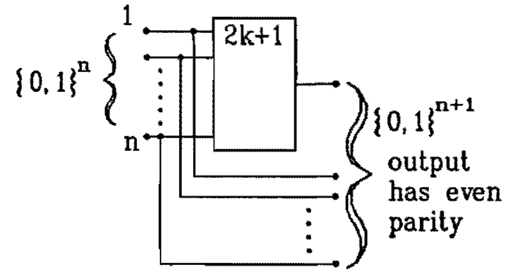
Figure 5.59 shows how to realize an 8-bit adder as an 8-iterative circuit of full adders.

When storing and transferring binary n-tuples sometimes errors can happen, disturbances occur. One or more elements of the n-tuple can get another value.

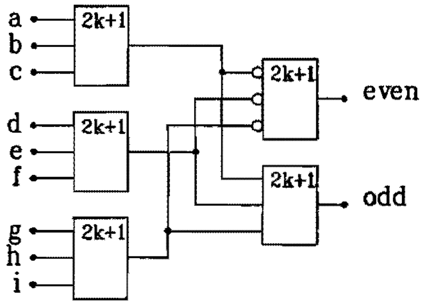
Frequently, an extra bit is added to such an n-tuple to be able to detect whether an error occurs. The value of this extra bit is chosen such that the total number of elements in the (n+1)-tuple that have the value 1 are even (or odd). We are then considering an even (or odd) *parity.* The question of odd or even parity (=number of elements that have the value 1) of an n-tuple, can be answered using an n-input exclusive-or function.

PARITY GENERATOR

- Parity generator

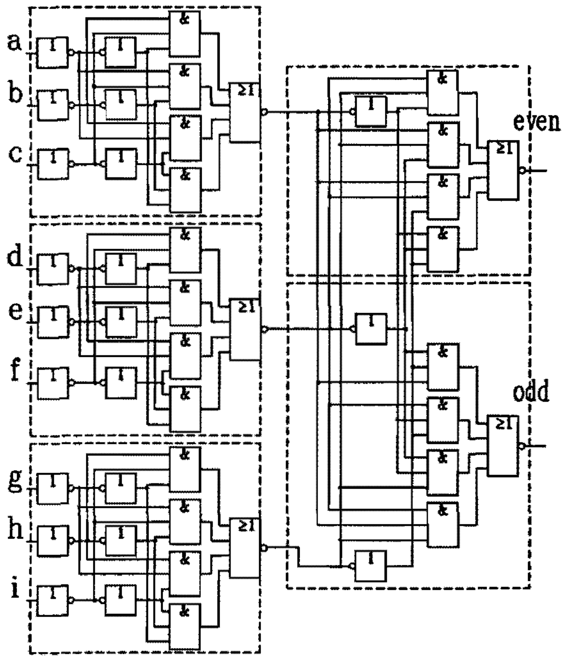


- 74280: 9-input parity generator/checker



-5.60-

SN74S280 PARITY GENERATOR/CHECKER



Source: Texas Instruments TTL Devices

-5.61-

Indeed, this function has the value 1 if an odd number of inputs have the value 1, i.e. the n-tuple has an odd parity. We see in figure 5.60 how we can make an (n+1)-tuple with an even parity from an n-tuple using an n-input exclusive-or. Also we have in figure 5.60 the principle schematic of the TTL 74280 building block. This is a 9-input parity generator/checker. We see that the necessary 9-input exclusive-or function is realized using a 2-layer tree structure with three input exclusive-or gates. We can simply say that an exclusive-or gate with inverted inputs is equal to an exclusive-or gate with an inverted output.

If we invert the inputs of an exclusive-or gate, we actually determine whether the number of zeroes at the input is odd.

Example: 001 # zeroes = 2 (even) = not odd
# ones = 1 (odd)

So inverting the inputs inverts the xor function; but this only holds if there is an odd number of inputs!

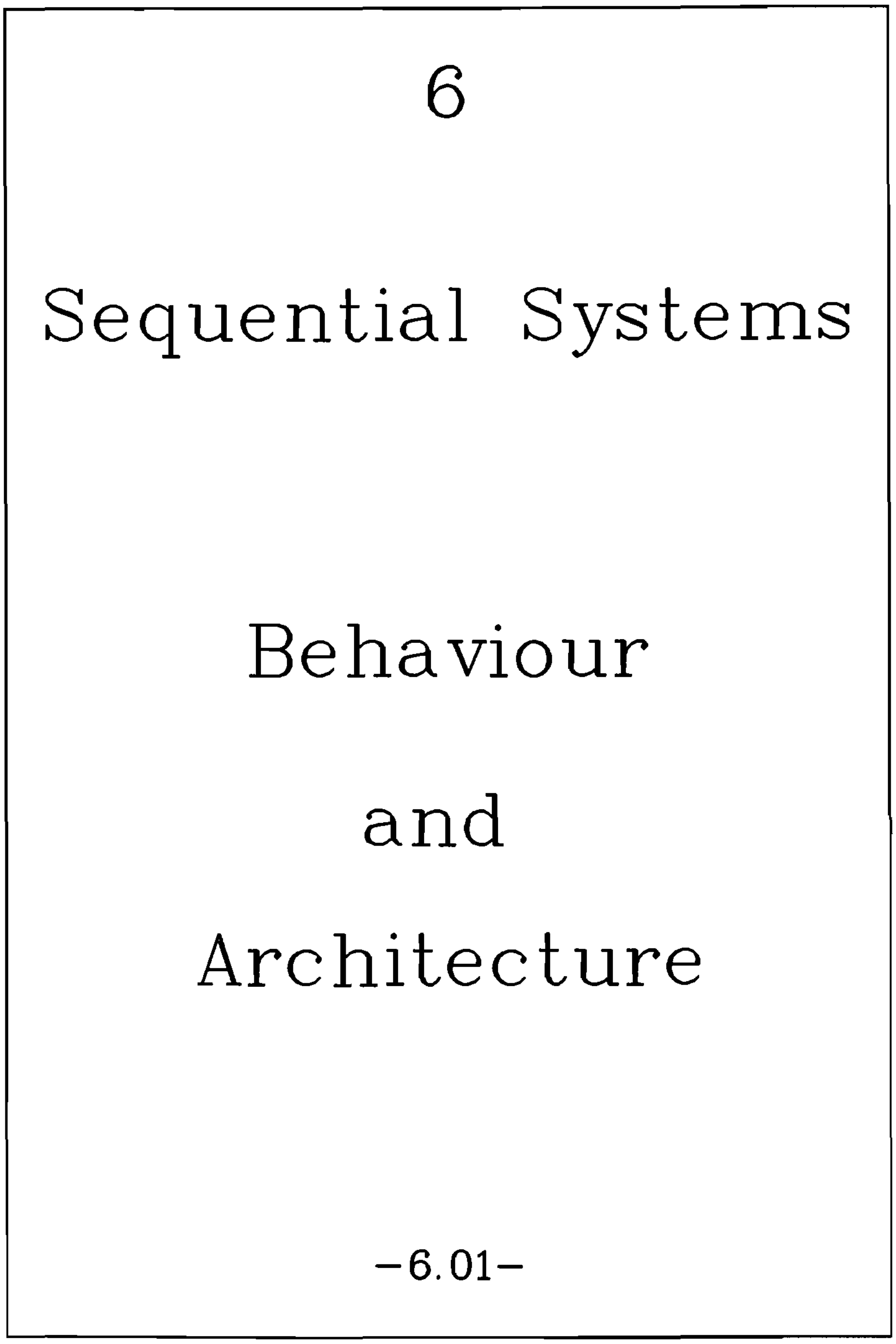
In figure 5.61 we find the SN74S280 schematic, which is an edited version of the schematic in a Texas Instruments data book. In this schematic we have shown the 3-input exclusive-or functions with a dotted line. We see that these functions are realized in the sum-of-minterms form. We also notice that a direct implementation of a 9-input exclusive-or function in a sum-of-minterms form will require a total of 256 and-gates. A tree realization is significantly advantageous.

5.7 Summary

In this chapter we have discussed the realization possibilities of combinational binary functions. Because all combinational binary functions have a sum-of-minterms form, a realization using and-gates and or-gates is always possible. The realization of a sum-of-minterms form can also be done using a programmable building block, the read-only memory or ROM. We have seen that in many cases the sum-of-minterms form can be converted into a sum-of-products (SOP) form, that can generally be realized with less gates and thus is cheaper. Searching for the SOP form with the smallest number of product-terms is called minimization. Programs are available for this. A minimal sum-of-products form leads to a cost-effective 2-layer and-or realization. Here also we can use programmable building blocks. We have discussed the PLA and the PAL. We have also seen that we cannot or do not want to always specify the value of the function for all the possible input combinations. We then speak of an *incompletely specified function*. For specific input combinations the function value is a don't care. Minimization procedures make use of these extra degrees of freedom to find an optimal realization. We emphasize again that we have occupied ourselves with the sum-of-minterms form and the sum-of-products form in chapter 5. For each combinational binary function there also exists a product-of-maxterms form and several product-of-sums forms. These lead to the dual or-and realization.

Furthermore, we discussed the realization of functions by exclusively using nand-gates or nor-gates, instead of and-gates and or-gates. Both gate types are simpler and cheaper to implement with electronic devices than the and-gates and or-gates. Due to the availability of programmable building blocks, such as PLAs and PALs, the use of the multiplexer as a universal building block is pushed into the background. Actually, as we have seen, the multiplexer offers good possibilities for the realization of general combinational binary functions. Finally, in this chapter we have introduced a fifth type of gate: the exclusive-or gate. The exclusive-or function plays an important role in the addition of two binary coded numbers, and in the detection and correction of errors. We have shown this in a number of examples.

In the previous chapters we have discussed how we can specify combinational systems, i.e. systems without memory. Along the way of systematic hierarchical design methodology we can divide such a system into a number of smaller subsystems. If these subsystems are sufficiently simple we can map them onto an equivalent combinational binary system via a coding step. The behaviour of such a system can be described in a truth table or a function table or by means of switching algebra expressions. Finally, we can realize the resulting switching functions using standard building blocks, such as the and-gate and the or-gate, or using programmable building blocks. Herewith we have completely run the path from specification to realization for combinational systems. We finally concluded that obviously any memoryless digital system is in principle realizable.



6

Sequential Systems

Behaviour

and

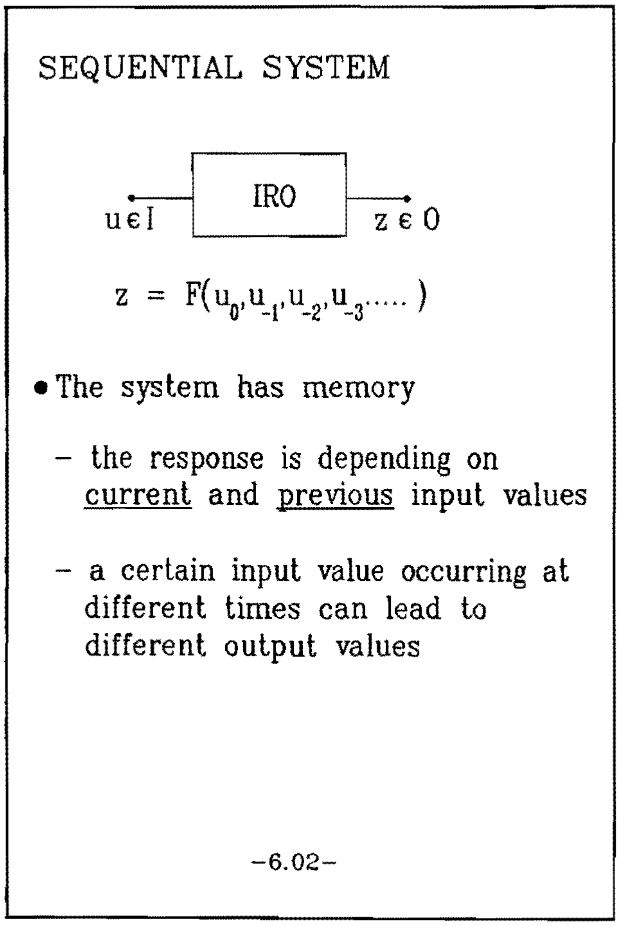
Architecture

-6.01-

6 Sequential systems

In chapter two we have already stated that, next to combinational systems, sequential systems play an important role. Both types of system differ from each other by the fact that in combinational systems the past plays no role; in sequential systems, however, it certainly does.

Sequential systems are not characterized by a simple functional relation between momentary values on the input and output. Of course, input and output values are related to each other, but previous input values are essential in this respect.

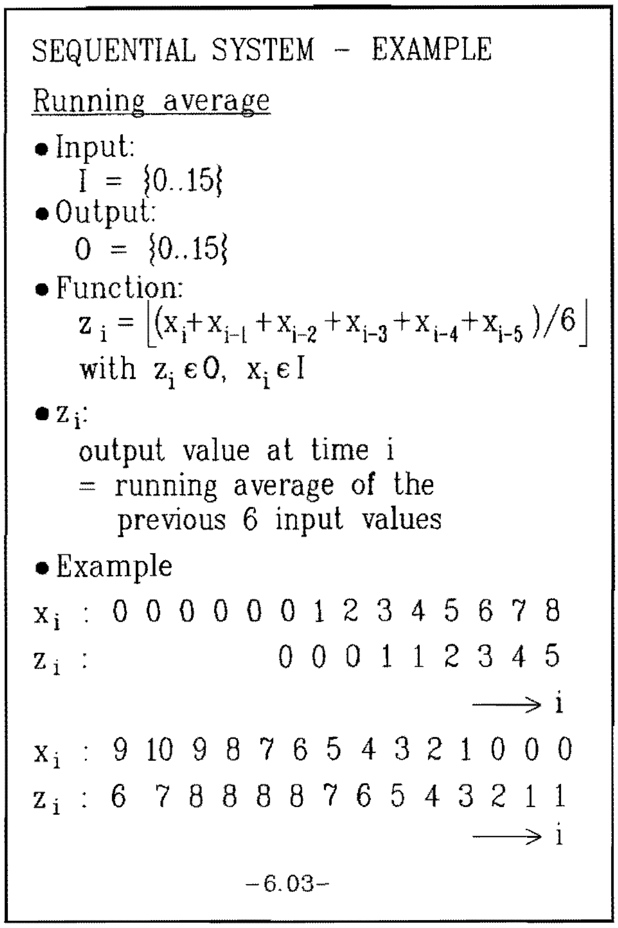


SEQUENTIAL SYSTEM

$z = F(u_0, u_{-1}, u_{-2}, u_{-3} \ldots\ldots)$

- The system has memory
  - the response is depending on <u>current</u> and <u>previous</u> input values
  - a certain input value occurring at different times can lead to different output values

-6.02-



SEQUENTIAL SYSTEM - EXAMPLE

<u>Running average</u>

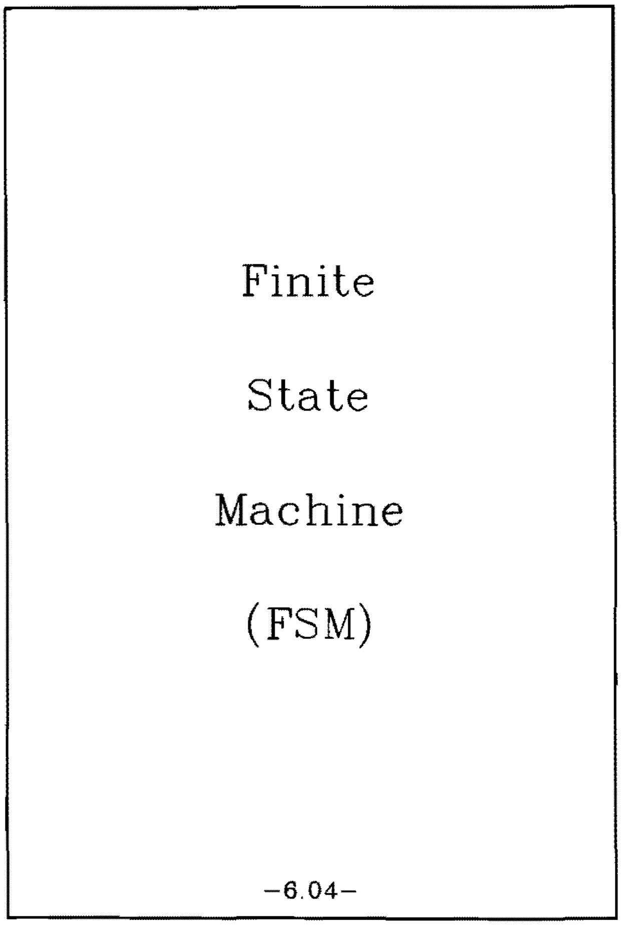
- Input:
  $I = \{0..15\}$
- Output:
  $0 = \{0..15\}$
- Function:
  $z_i = \lfloor (x_i + x_{i-1} + x_{i-2} + x_{i-3} + x_{i-4} + x_{i-5})/6 \rfloor$
  with $z_i \in 0$, $x_i \in I$
- z_i:
  output value at time i
  = running average of the previous 6 input values
- Example

```
xi : 0 0 0 0 0 0 1 2 3 4 5 6 7 8
zi :           0 0 0 1 1 2 3 4 5
                             ——> i
xi : 9 10 9 8 7 6 5 4 3 2 1 0 0 0
zi : 6  7 8 8 8 8 7 6 5 4 3 2 1 1
                             ——> i
```

-6.03-

We say (figure 6.02) that the sequential system has *memory functions.* The momentary value on the output is depending on the current and previous input values. The system keeps track of what the previous input values were. An important consequence is that in a sequential system a certain input value can lead to several different output values at different times. In chapter two we said that the repeated selection of the same digit of a telephone number leads to different reactions in a telephone exchange, being a large sequential system.

Figure 6.03 shows another example of a sequential system. Let the domain of the input and output values be defined by the integer numbers from 0 to 15. We can then determine the running average of the last 6 input values by adding the current input value to the 5 preceding ones, and dividing the whole by 6. We must still round the result to an integer number. Notice that the system must save 6 input values to be able to compute the momentary output value. In figure 6.03 we also show the output process as a response to the given process of the input values. We begin first with the introduction of 6 zeroes. Notice that we only know after the sixth zero what the output will be. The following output values can be simply calculated from the given input values. Notice that here we are clearly considering a sequential system. The output z_i is not only depending on the current input value but also on the previous ones. So it can happen that a certain input value can lead to different output values. Notice that the time - in the sense that some actions occur before others - now plays an important role. The order in which input values are introduced determines the output value.



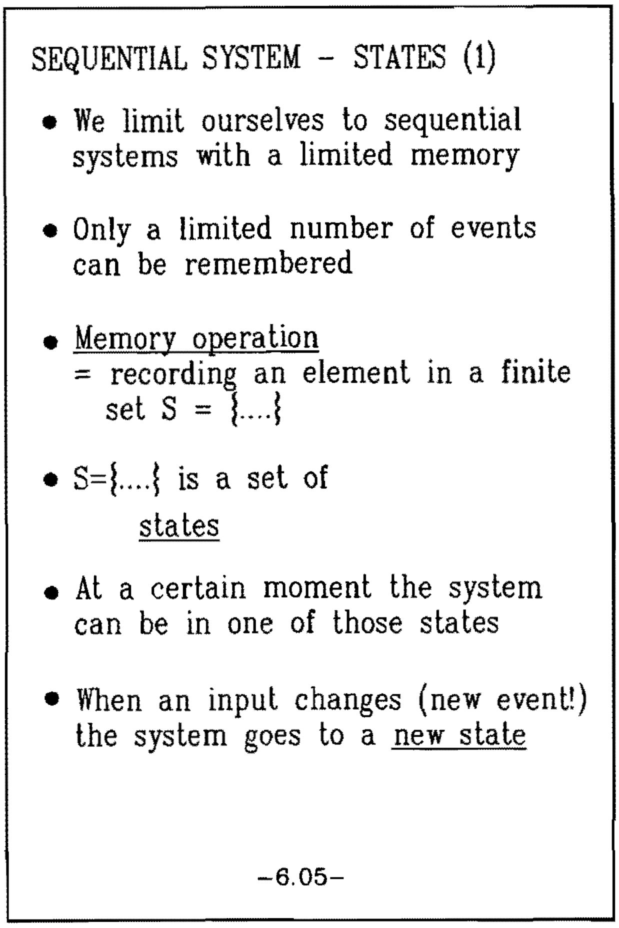
Finite

State

Machine

(FSM)

–6.04–



SEQUENTIAL SYSTEM – STATES (1)

- We limit ourselves to sequential systems with a limited memory
- Only a limited number of events can be remembered
- Memory operation
  = recording an element in a finite set S = {....}
- S={....} is a set of states
- At a certain moment the system can be in one of those states
- When an input changes (new event!) the system goes to a new state

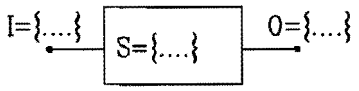
–6.05–

6.1 Finite State Machine (FSM)

Each digital system for which the output value is determined by the past input values, is a sequential system. This is somewhat problematic: although the value set of the inputs and outputs have a finite number of elements, this does not say anything about the relation between the input and output values. A specially interesting question is how much of the past we must remember to determine the new output values. In a number of cases it appears that this amount of remembered information is so large that the specification and realization of the sequential system becomes a problem.

To avoid this sort of problem, we limit ourselves in this course to an important subclass of sequential systems, viz. those for which the memory has a limited size, i.e. only a limited number of items can be stored (see figure 6.05). More precisely, we formalize the memory function as the saving of an element belonging to a finite set. An event which we have to remember can be specified by an element of this finite set S. The elements of the S set are called the *system states*. When a special input value (a new *event*) occurs, the system assumes a next state. The fact that the system is in that next state, is a result of the occurring event, reflecting that the corresponding input value has occurred.

SEQUENTIAL SYSTEMS – STATES (2)

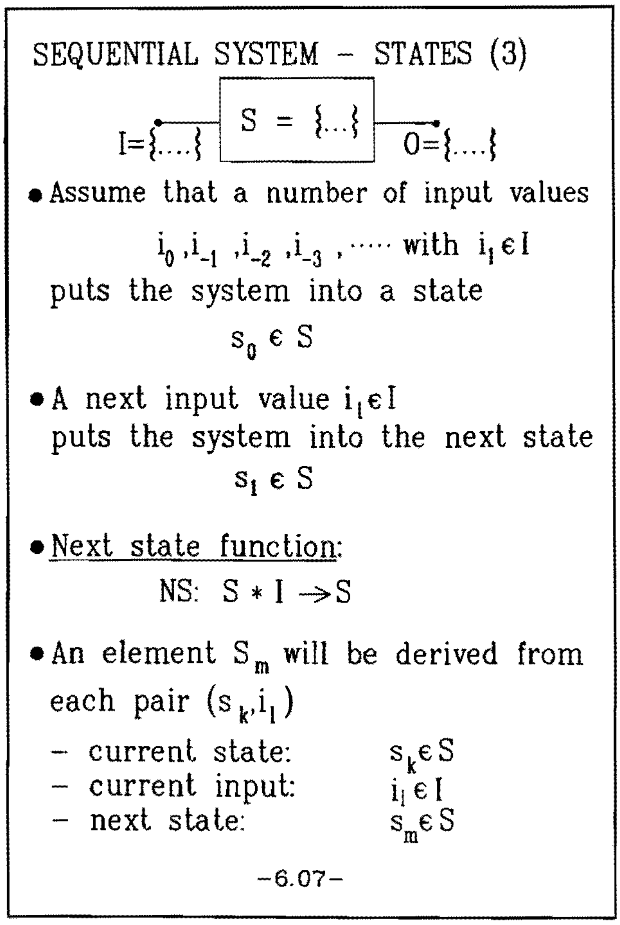


- Assume that a number of input values $a_0, a_{-1}, a_{-2}, a_{-3}, a_{-4}, ..$ with $a_i \in I$ puts the system into a state $s_0 \in S$
- Assume that another series of input values $b_0, b_{-1}, b_{-2}, b_{-3}, b_{-4}, ..$ with $b_i \in I$ puts the system into the same state $s_0 \in S$
- This means the system cannot distinguish the two input series
- Responses to $c_1, a_0, a_{-1}, a_{-2}, a_{-3}, a_{-4}, ..$ with $c_1, a_i \in I$ and $c_1, b_0, b_{-1}, b_{-2}, b_{-3}, b_{-4}, ..$ with $c_1, b_i \in I$ are identical
- The past is included in the current state of the system

–6.06–

The sequential systems that we approach in this course are thus defined as three finite value sets. In figure 6.06 these are expressed as I for the set of input values, S for the set of states, and O for the set of output values. The present and previous input values determine the system's current state. We can also say that a time series of input values puts the system into a determined state (see figure 6.06). With a limited number of different input values (finite value set) we can make a large number of different series of input values.

Check for yourself, for example, in how many different ways we can make a series of ten digits from the digits 0 to 9. Because our system only has a finite number of states each series of inputs values cannot put the system in yet another new state, as there are too few of them. Many different sequences of input values will put the system in the same state as s_0 (see figure 6.06). The system contains insufficient input states to distinguish the series a_0.... and b_0.... The system does not know which of the series a_0.... or b_0.... are responsible for the current state s_0. Consequently, if for both sequences the following value is c_1, the system produces an identical response. We say that the past, the series of input values that has occurred is completely taken into account in the current system state. Only the current state determines what the system's response will be upon a following input value.



SEQUENTIAL SYSTEM - STATES (3)

I={....} S = {...} O={....}

- Assume that a number of input values

  $i_0, i_{-1}, i_{-2}, i_{-3}, \dots$ with $i_l \in I$

  puts the system into a state

  $s_0 \in S$

- A next input value $i_l \in I$

  puts the system into the next state

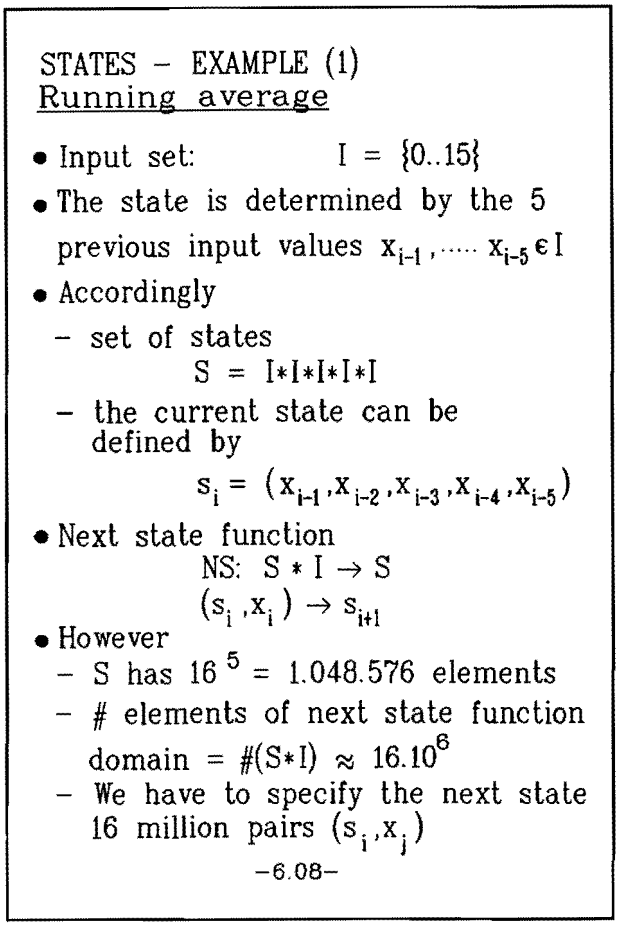
  $s_l \in S$

- Next state function:

  NS: $S * I \rightarrow S$

- An element S_m will be derived from each pair (s_k, i_l)
  - current state: $s_k \in S$
  - current input: $i_l \in I$
  - next state: $s_m \in S$

-6.07-



STATES - EXAMPLE (1)

Running average

- Input set: $I = \{0..15\}$
- The state is determined by the 5 previous input values $x_{i-1}, \dots x_{i-5} \in I$
- Accordingly
  - set of states

    $S = I*I*I*I*I$
  - the current state can be defined by

    $s_i = (x_{i-1}, x_{i-2}, x_{i-3}, x_{i-4}, x_{i-5})$
- Next state function

  NS: $S * I \rightarrow S$

  $(s_i, x_i) \rightarrow s_{i+1}$
- However
  - S has 16^5 = 1.048.576 elements
  - # elements of next state function domain = $\#(S*I) \approx 16.10^6$
  - We have to specify the next state 16 million pairs (s_i, x_j)

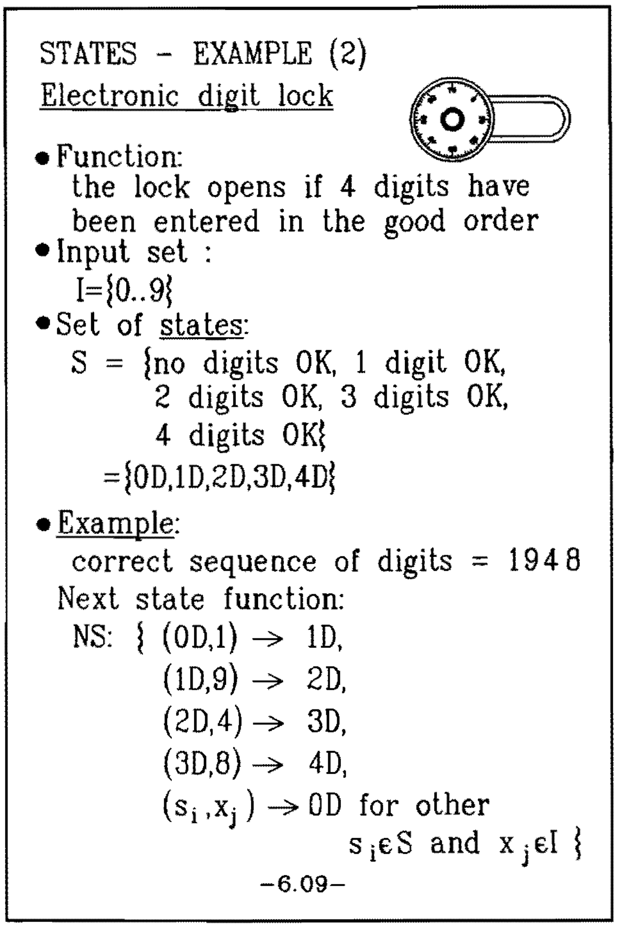
-6.08-

In figure 6.07 we have summarized this behaviour of our sequential system again. We shall call such a system a *finite state machine*. As we have seen, a sequence of input values puts the system in a specific state. The following input value then puts the system in a next state. This following state is depending on the momentary input value i_1 and the past, that is taken into account in the current state s_0. We shall exclusively discuss our approach for finite state machines for which the next state is given as a function of the current state and the current input value. Thus, we are considering the next state function NS. This function relates an element s_m to each pair (s_k, i_l) introducing the following system state. This following state function determines the behaviour of our sequential system. Furthermore, it is not always necessary that the new state s_m has to be different form the old state s_k. In that case the machine's state does not change under the influence of the new input value i_l.

Consider, as an example, the sequential system for determining the running average.

In figure 6.08 we have again expressed that the set of input values comprises the integer numbers from 0 to 15. First we ask ourselves what is the set of states for the system. We know that the system must save the previous 5 input values to be able to determine the running average.

It is convenient to denote the momentary system state by this 5-tuple. The total set of states is then formed by all 5 digits, where the digits have a value from 0 to 15. If we want to describe the behaviour of our system, we shall have to begin with the specification of the next state function NS. Here we really have a problem. As shown the state set S has in total 16^5=1,048,576 elements and the domain of the next state function has itself about 16 million elements. This means that we need to specify the following state for 16 million pairs (s_i, x_j). This is an impossible exercise. We cannot specify such a machine in this way. Thus, the conclusion must be: it is far from true that any sequential digital system is describable and thus realizable as a finite state machine. Obviously we need to have other methods to design these types of machines. We shall return to this in the following chapter.



STATES - EXAMPLE (2)

Electronic digit lock

- Function:
  the lock opens if 4 digits have been entered in the good order
- Input set :
  $I=\{0..9\}$
- Set of states:
  $S = \{$no digits OK, 1 digit OK, 2 digits OK, 3 digits OK, 4 digits OK$\}$
  $=\{0D,1D,2D,3D,4D\}$
- Example:
  correct sequence of digits = 1948
  Next state function:

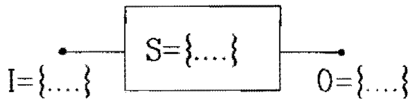
$$
\begin{aligned}
NS: \{\ & (0D,1) \rightarrow 1D, \\
& (1D,9) \rightarrow 2D, \\
& (2D,4) \rightarrow 3D, \\
& (3D,8) \rightarrow 4D, \\
& (s_i,x_j) \rightarrow 0D \text{ for other} \\
& \qquad s_i \epsilon S \text{ and } x_j \epsilon I\ \}
\end{aligned}
$$

-6.09-

As an example of a systems that is realizable as a finite state machine we identify the electronic digital lock. In figure 6.09 we go further into the function of this digital lock. The point is that the lock may only open if 4 correct digits are input sequentially. If a wrong digit occurs in a series of 4 digits, the system returns back to the state where no correct digits were input. We see that the set of input values I consists of the digits from 0 to 9, and the set of output values consists of the closed or open state of the lock. We indicate how many correct digits are input with the system's possible states.

The set of states S is thus given by: "no correct digit", "one correct digit", "two correct digits", "three correct digits" and "four correct digits". We abbreviate this with the symbols 0D to 4D. If 1948 is the correct sequence of input digits, we can specify the next state function as shown in figure 6.09. Initially we are in the state 0D; no correct digit is received. Only the input of the digit 1 puts us into the state 1D. The input of any other digits does not cause any state change. In state 1D the input of the digit 9 puts the system in state 2D: two correct digits have been recorded. All other digits take us back to state 0D. You can check the correctness of the given specification for yourself.

NEXT OUTPUT FUNCTION

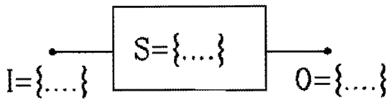


- The new output value $z_i \epsilon O$ is determined by the current and previous input values $x_i, x_{i-1}, x_{i-2}, ... \epsilon I$
- Previous input values (the past) are (is) taken into account in the current state $s_i \epsilon S$
- Next Output function
  NO: $S * I \rightarrow O$
- An element Z_m will be related to each pair (s_k, i_l)
  - current state: $s_k \epsilon S$
  - current input: $i_l \epsilon I$
  - next output value: $z_m \epsilon O$
- <u>Example</u>: digit lock
  NO: { (3D,8) → open,
  (s_i, x_j) → locked for other $(s_i, x_j) \epsilon S*I$

-6.10-

FINITE STATE MACHINE

- Sequential system



- Input set I with limited number of elements
- Output set O with limited number of elements
- State set S with limited number of elements
- Next State function NS: $S * I \rightarrow S$
- Next Output function NO: $S * I \rightarrow O$

- <u>Finite State Machine (FSM)</u>
  defined by 5-tuple

FSM = (I,S,O,NO,NS)

-6.11-

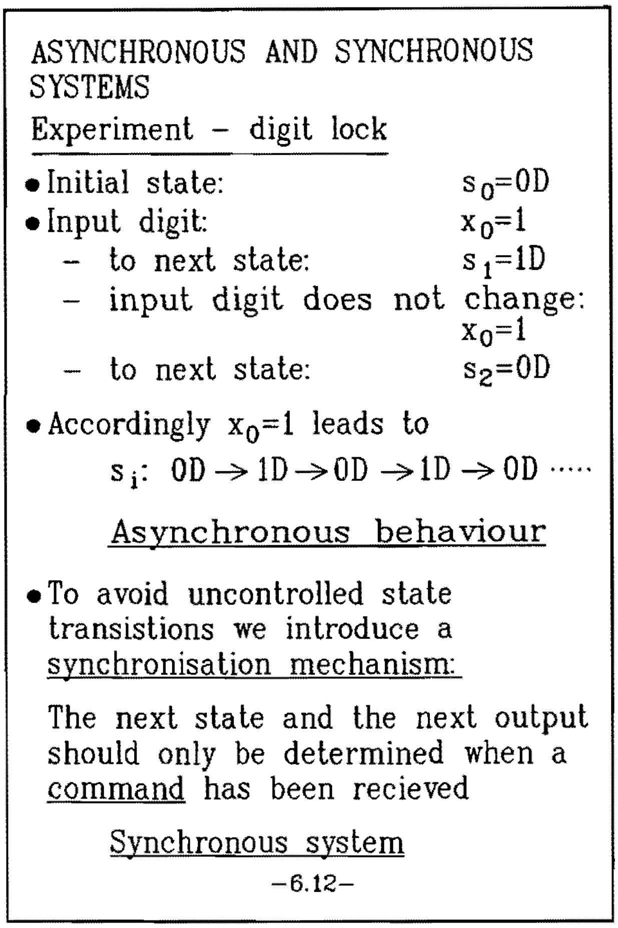
In the behaviour of sequential systems, states are not the only important issue. The output values are even more important. These output values that change in response to the changing input values are what we, the users of the sequential system, receive and observe.

In general we can say (see figure 6.10) that the new value z_i on the output is determined by the new input value and the previous input values. Thus, the past plays a role here. This past is taken into account in the system's current state. Thus, from this current state together with the new input value a new output values is determined. We limit ourselves to systems for which this relation is given by a function, the Next Output function NO. This function, representing the new output value of the system, relates an element z_m to each pair (s_k, i_l). As an example, figure 6.10 shows the next output function, NO, of our digital lock. This function is very simple. Only if 3 correct digits have been input (i.e. we are in state 3D), then the lock will be opened after the input of the digit 8. Thus, the current state, 3D, together with the new input 8 produces the new output "open". In all other cases the new output is equal to "locked".

Summarizing, we see in figure 6.11 that we limit ourselves, in relation to sequential systems in this course, to systems which can be described by the following 5 items:

- An input set I with a finite number of elements
- An output set O with a finite number of elements
- A state set S with a finite number of elements
- A next state function NS describing the relation between input and state
- A next output function NO describing the output value

We call such a system a *finite state machine*, an FSM. Such an FSM is defined by the 5-tuple (I, S, O, NO, NS). We note that the theory of finite state machines goes further than the scope of this course.



ASYNCHRONOUS AND SYNCHRONOUS SYSTEMS

Experiment – digit lock

- Initial state: s_0=0D
- Input digit: x_0=1
  - to next state: s_1=1D
  - input digit does not change: x_0=1
  - to next state: s_2=0D
- Accordingly x_0=1 leads to

  s_i: 0D -> 1D -> 0D -> 1D -> 0D .....

  Asynchronous behaviour

- To avoid uncontrolled state transistions we introduce a synchronisation mechanism:

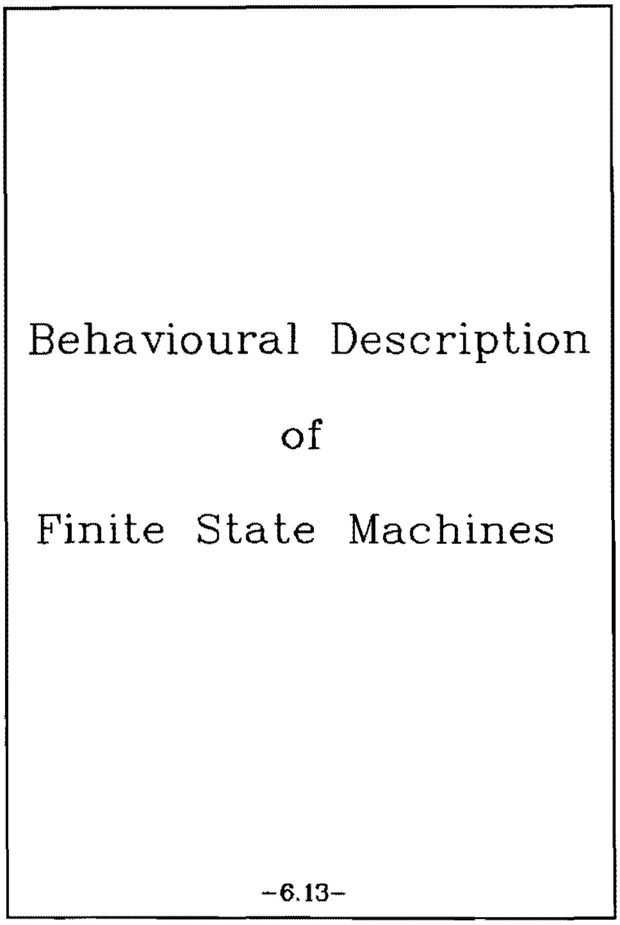
  The next state and the next output should only be determined when a command has been recieved

  Synchronous system

–6.12–

At the end of this paragraph we shall perform a small mental experiment. Let us consider the electronic digit lock (figure 6.12). Assume that the system is in the initial state s_0 = 0D, i.e. no correct digits have been input. If we now put the digit 1 on the input, then we know that the system will go to the following state s_1 = 1D. Arriving at this state, the value of the input will in general be unchanged and thus still equal to the digit 1. Actually, the next state function shows for this situation (=input value + state) that the following state must be 0D. Thus, the system goes to its next state s_2 = 0D. We see that we get, as a result, a system that remains alternating (bouncing) between both states, 0D and 1D. The system does not arrive at a stable state. Systems that immediately react on inputs are called *asynchronous systems.*

Obviously the manner in which we specify the next state function is not suitable for the definition of that sort of asynchronous systems. In this course we shall limit ourselves to *synchronous* systems. Such systems have a synchronization mechanism. This mechanism ensures that after each reception of an instruction the system only determines the following state and following output one time. Applied to our digit lock, this means the following: In the initial state s_0 = 0D we set the digit 1 on the input. We then give the system the instruction to go to the next state. The system thus goes to a state s_1 = 1D and waits there for an instruction to go to the following state. Now we have the time to change the value on the input before we supply this new instruction. Later we shall see how we can simply implement this type of system.

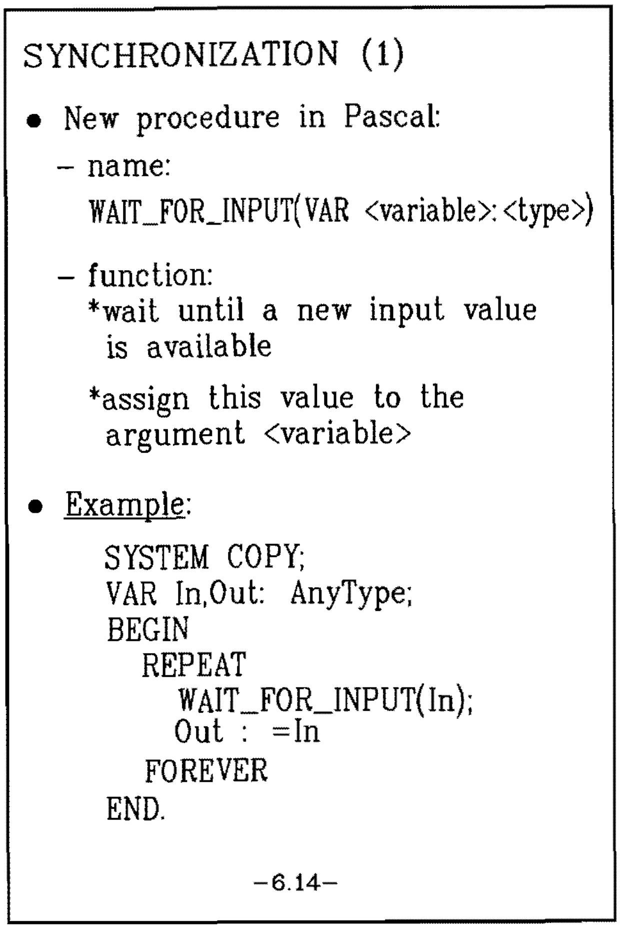


Behavioural Description

of

Finite State Machines

-6.13-



SYNCHRONIZATION (1)

- New procedure in Pascal:
  - name:
    WAIT_FOR_INPUT(VAR <variable>: <type>)
  - function:
    *wait until a new input value is available
    *assign this value to the argument <variable>
- Example:

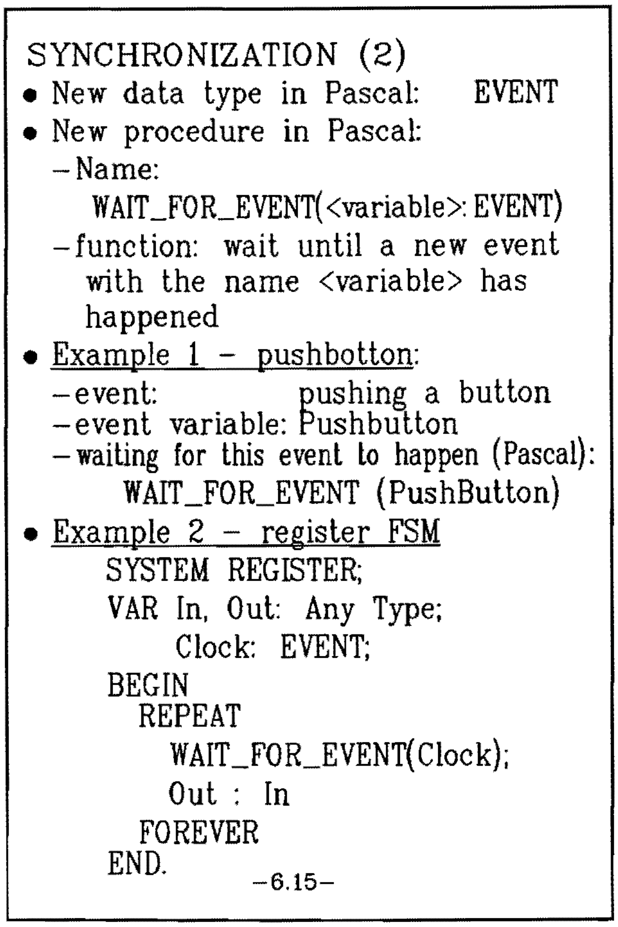
```
SYSTEM COPY;
VAR In,Out:  AnyType;
BEGIN
  REPEAT
    WAIT_FOR_INPUT(In);
    Out :  =In
  FOREVER
END.
```

-6.14-

6.2 Behavioural description of finite state machines

Also in the design of sequential machines we make use of the structured hierarchical design methodology. That is to say, we have to ask ourselves (in the case of sequential systems): "What must our system do?", and only after that we look for the answer of the question: "how do we realize such a system?". Thus we must concern ourselves with the behavioural description, i.e. the specification of our sequential system. Here we limit ourselves, as mentioned earlier, to finite state machines. The fact that time plays an important role in the behaviour of our sequential systems makes the behavioural description more complex. In programming languages such as the language used in this course (Pascal), we need language constructs to show that the system must wait for the external occurrence of commands. The next state and output can only be set after receiving such a command. We shall include two facilities specially for this. First, we assume the availability of a new standard procedure: WAIT_FOR_INPUT.

If we use this standard procedure in a behavioural description, we want to indicate that in this place the system must wait until a new input value is available. This new input value is then assigned to the arguments given to the standard procedure. As an example of the use of this standard procedure, figure 6.14 shows a description of a system that only copies the input value to the output. With the REPEAT FOREVER construct we show that we here are considering sequential system behaviour. The description is continually repeated in time. What is actually repeated infinitely?
We see first the WAIT_FOR_INPUT procedure, where we wait until a following input value is available. This new value is then assigned to the variable In, and in the following statement copied to the output. Here the system is going to wait again for the following input value.



SYNCHRONIZATION (2)

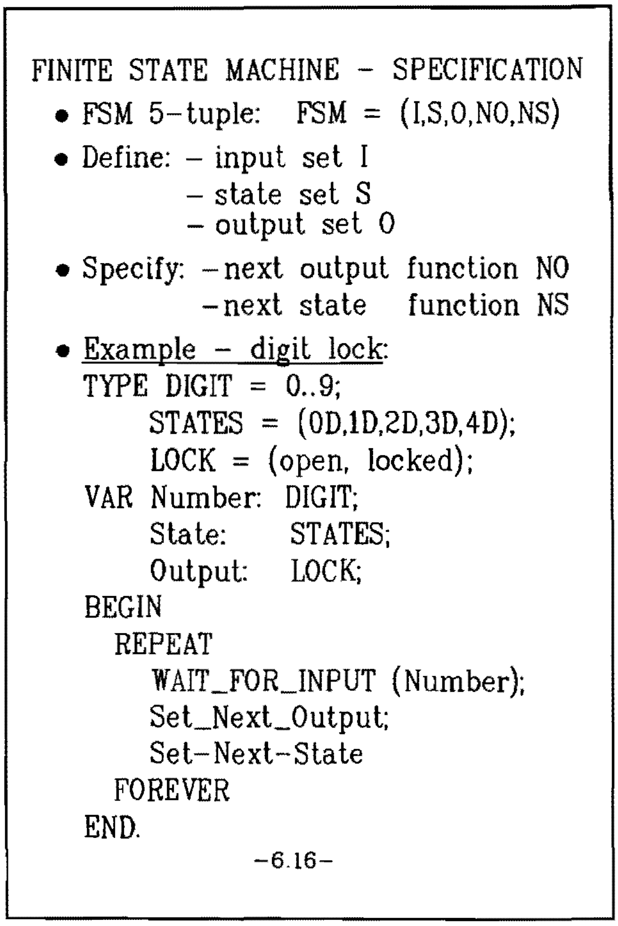
- New data type in Pascal: EVENT
- New procedure in Pascal:
  - –Name:
    WAIT_FOR_EVENT(<variable>: EVENT)
  - –function: wait until a new event with the name <variable> has happened
- <u>Example 1 - pushbotton</u>:
  - –event: pushing a button
  - –event variable: Pushbutton
  - –waiting for this event to happen (Pascal):
    WAIT_FOR_EVENT (PushButton)
- <u>Example 2 - register FSM</u>

```
SYSTEM REGISTER;
VAR In, Out: Any Type;
      Clock: EVENT;
BEGIN
  REPEAT
    WAIT_FOR_EVENT(Clock);
    Out : In
  FOREVER
END.
```

–6.15–

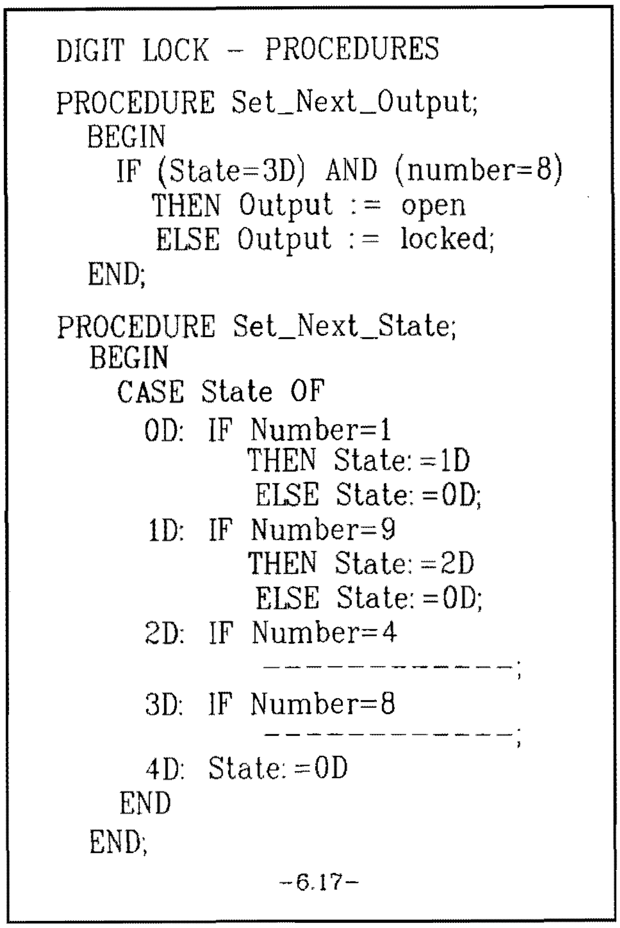
Sometimes we also want the sequential system to wait for external events, such as a push-button or the interruption of a light beam. Here the input value is not interesting; the fact that the event happened is much more important. To describe such a situation in our behavioural description, we need a new standard type (see figure 6.15). We shall call the new standard type an EVENT. Thus we can have in our description variables of the type event. We add here an extra standard procedure, the WAIT_FOR_EVENT procedure. With this procedure we express that we want to wait until a new event, corresponding to the variable name, has occurred.

A variable with the name "button" can indicate the event "the button will be pushed". Figure 6.15 shows how we can simply describe a *register* using this mechanism. A register is a very simple finite state machine that does not do anything except, when given a command, saves the input value in its memory and in the same time puts this value on its output. The command is given via a special input which is frequently call *clocked input.* Accordingly, we have defined in our description a variable clock of the type EVENT. The description is very simple. We wait for the occurrence of the clock event, and then we copy the input value In to the variable Out, and it is kept until the next clock event. Notice that we shall not further specify or use the value of the clock, whatever it may be. This value is not interesting from the viewpoint of our system behaviour.



```
FINITE STATE MACHINE - SPECIFICATION
  • FSM 5-tuple:   FSM = (I,S,O,NO,NS)
  • Define: - input set I
            - state set S
            - output set O
  • Specify: -next output function NO
             -next state  function NS
  • Example - digit lock:
    TYPE DIGIT = 0..9;
         STATES = (0D,1D,2D,3D,4D);
         LOCK = (open, locked);
    VAR Number: DIGIT;
         State:    STATES;
         Output:   LOCK;
    BEGIN
      REPEAT
         WAIT_FOR_INPUT (Number);
         Set_Next_Output;
         Set-Next-State
      FOREVER
    END.
```

-6.16-



```
DIGIT LOCK - PROCEDURES
PROCEDURE Set_Next_Output;
  BEGIN
    IF (State=3D) AND (number=8)
       THEN Output := open
       ELSE Output := locked;
  END;
PROCEDURE Set_Next_State;
  BEGIN
    CASE State OF
      0D: IF Number=1
             THEN State:=1D
              ELSE State:=0D;
       1D: IF Number=9
             THEN State:=2D
              ELSE State:=0D;
      2D: IF Number=4
              ------------;
      3D: IF Number=8
              ------------;
      4D: State:=0D
    END
  END;
```

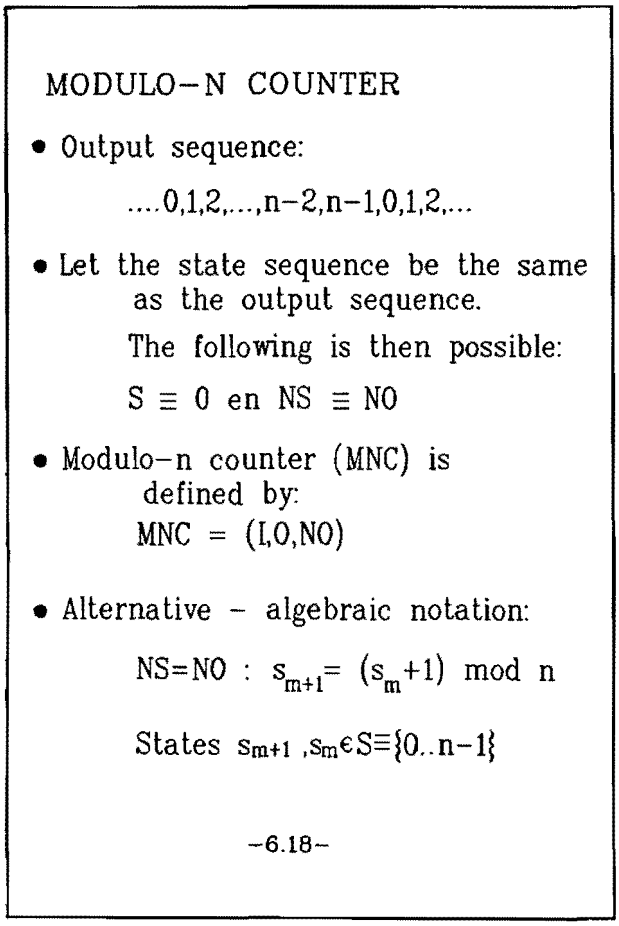
-6.17-

Now we know how to define synchronization with the external world in our description. We can now proceed to the description of the behaviour of various finite state machines. In figure 6.16 we show what is needed to determine this behaviour. First we see that the input value set, the output, and the states must be defined. When we have done that we can specify the next output (NO) and the next state (NS) functions. We shall explain them both using a number of examples.

As a first example we show in figure 6.16 the behaviour of our electronic digit lock. This behavioural description begins with the definition of the input set, the state set and the output set, using type declarations. Subsequently we include the declaration of the input variable, the state variable and the output variable. The core of the behavioural description is very simple. We first wait for a new input digit. Then we determine the following output value and the following state. In the behavioural description we have shown this using two procedures: Set_Next_Output, and Set_Next_State. We notice also that a language, such as Pascal, is sequential. The sequential writing of both procedure calls suggests that both procedures are executed after each other. Thus, first calculating the following output and then determining the following state. The execution does not necessarily have to be sequential. In many cases the determination of the following output and the determination of the following state can be parallel, i.e. will take place in the same time. That is, we do not have to first determine the following state and then (afterwards) determine the next output. Indeed, for the determination of the new output value the current (momentary) state value is necessary. Figure 6.17 shows both procedures. Compare this behavioural description with the previously given functional specification for the next output function in figure 6.10 and for the next state function in figure 6.09. Notice that these behavioural descriptions are combinational functions, and that we do not need a REPEAT FOREVER construct. The given behavioural description are self-explanatory.

Problem 6.1 (6.1)

In the description of the Set_Next_State procedure (figure 6.17) the IF statements are not filled in for the cases "State=2D" and "State=3D". Complete this description by filling in these statements.



MODULO−N COUNTER

- Output sequence:

  ....0,1,2,...,n−2,n−1,0,1,2,...

- Let the state sequence be the same as the output sequence.

  The following is then possible:

  S ≡ 0 en NS ≡ NO

- Modulo−n counter (MNC) is defined by:

  MNC = (I,O,NO)

- Alternative − algebraic notation:

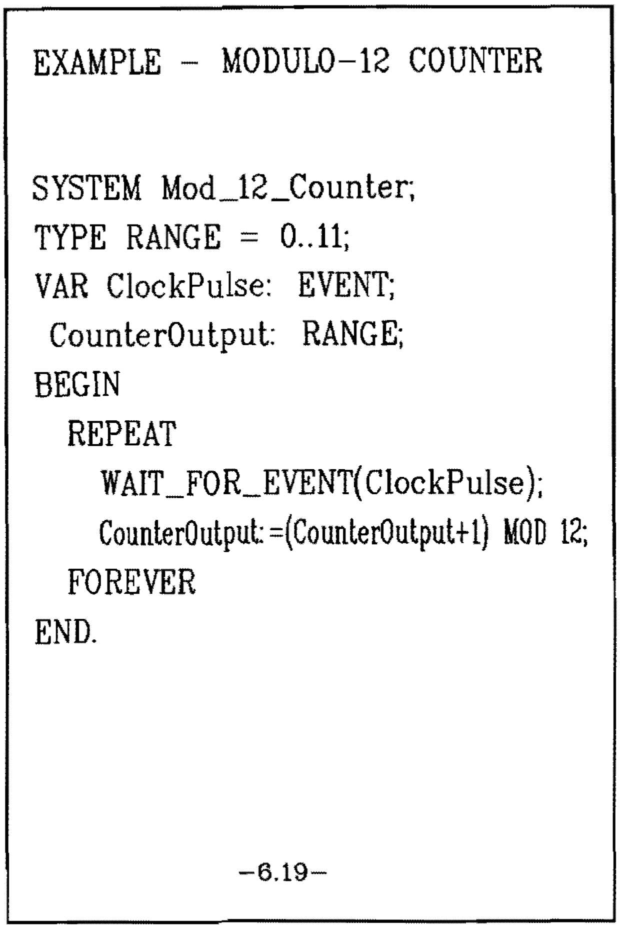
  NS=NO : $s_{m+1} = (s_m + 1) \bmod n$

  States $s_{m+1}, s_m \in S \equiv \{0..n-1\}$

−6.18−

A modulo-n counter is a sequential system that, upon a command of an external counting input, runs through a sequence of the integer numbers modulo-n. The system begins with 0, and then goes to 1, then 2, etc. to n-2 and finally to n-1, beginning again with 0. Such a system can be used, for example, to count the number of times that an external event occurs. We could count the number of visitors of an exibition, or the number of articles that pass an assembly-line belt. Such a modulo-n counter can be best realized by making the sequence of states equal to the sequence of the output values: 0, 1, 2 ... n-2, n-1, 0, etc. The set of states is thus equal to the set of output values. We also have to deal with the same function for the determination of the following state and the following output.

We see that a modulo-n counter is completely defined by the triple (I, O, NO). Furthermore, in this case we can also, as shown in figure 6.18, algebraically specify the next state and the next output function.



EXAMPLE − MODULO−12 COUNTER

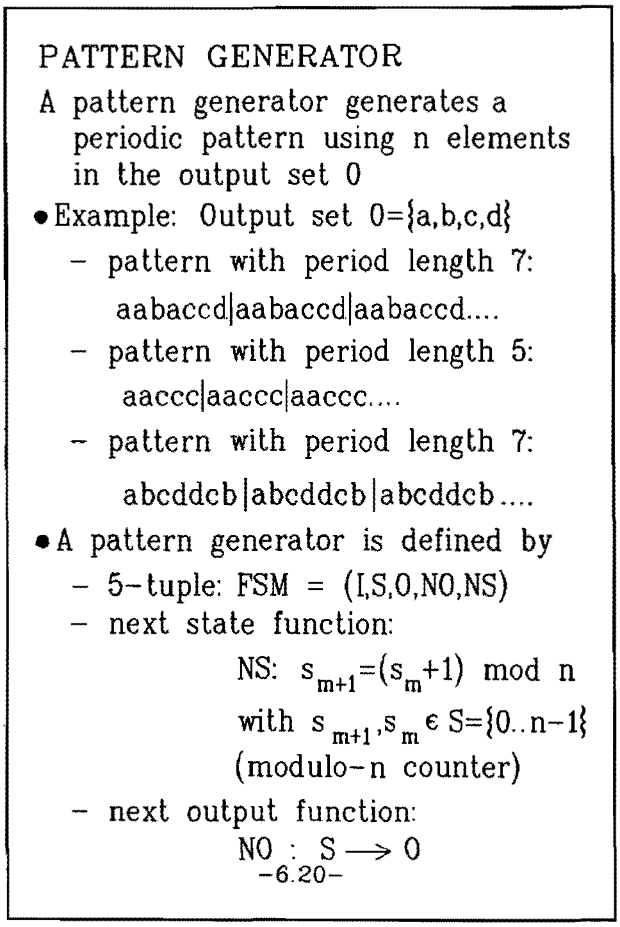
```
SYSTEM Mod_12_Counter;
TYPE RANGE = 0..11;
VAR ClockPulse: EVENT;
 CounterOutput: RANGE;
BEGIN
  REPEAT
    WAIT_FOR_EVENT(ClockPulse);
    CounterOutput:=(CounterOutput+1) MOD 12;
  FOREVER
END.
```

−6.19−

We shall make use of that later. In figure 6.19 we have specified the behaviour of a modulo-12 counter. In such a counter the range of the state set and, thus, the output set is from 0 to 11. Notice further that we define a variable ClockPulse, of type EVENT, to represent the external events. In the behavioural description we wait for the arrival of this counting pulse, after which the contents of the counter are increased by 1 modulo 12.

Remarks:
We have defined the modulo-n counter as a triple where the set of the input values I is included. This set is empty, i.e. the modulo-n counter has no ordinary inputs. Later we shall discuss modulo-n counters that can be set to a specific starting value, or that can be started or stopped by specific signals. In this case the set of input values I will not be empty.

Problem 6.2 (6.2)
Give a description of a modulo-5 counter. This counter contains, in addition to the counter pulse input, an ordinary input with a value set {start, stop}. After an occurring external event the counter will only assume its counting state if the value of this input is equal to "start".

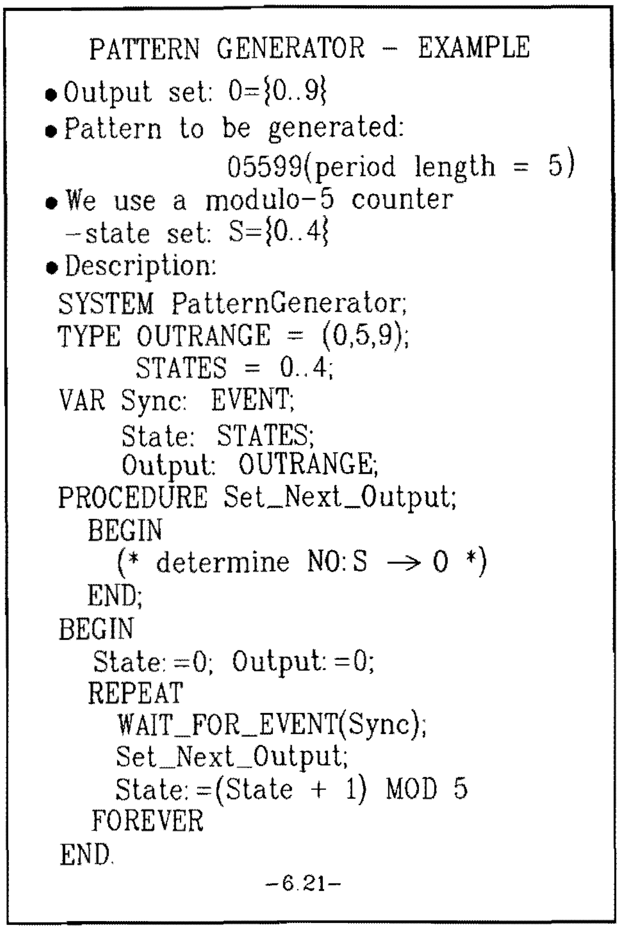


PATTERN GENERATOR

A pattern generator generates a periodic pattern using n elements in the output set 0

- Example: Output set 0={a,b,c,d}
  - pattern with period length 7:
    aabaccd|aabaccd|aabaccd....
  - pattern with period length 5:
    aaccc|aaccc|aaccc....
  - pattern with period length 7:
    abcddcb|abcddcb|abcddcb....
- A pattern generator is defined by
  - 5-tuple: FSM = (I,S,0,NO,NS)
  - next state function:
    NS: $s_{m+1}=(s_m+1) \bmod n$
    with $s_{m+1}, s_m \in S=\{0..n-1\}$
    (modulo-n counter)
  - next output function:
    NO : S → 0

-6.20-



PATTERN GENERATOR - EXAMPLE

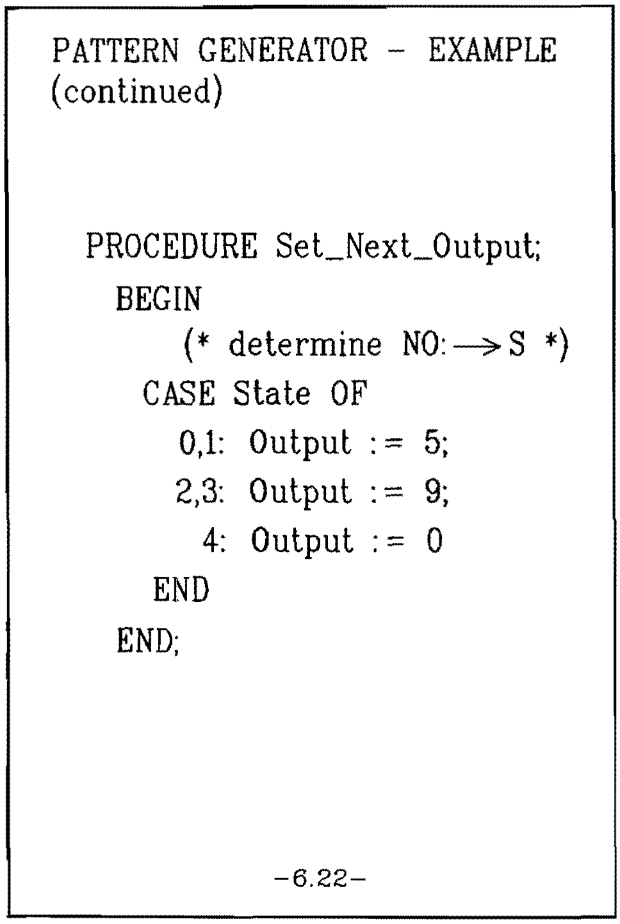
- Output set: 0={0..9}
- Pattern to be generated:
  05599(period length = 5)
- We use a modulo-5 counter
  -state set: S={0..4}
- Description:

```
SYSTEM PatternGenerator;
TYPE OUTRANGE = (0,5,9);
     STATES = 0..4;
VAR Sync: EVENT;
     State: STATES;
     Output: OUTRANGE;
PROCEDURE Set_Next_Output;
  BEGIN
    (* determine NO:S -> 0 *)
  END;
BEGIN
  State:=0; Output:=0;
  REPEAT
    WAIT_FOR_EVENT(Sync);
    Set_Next_Output;
    State:=(State + 1) MOD 5
  FOREVER
END.
```

-6.21-

A *pattern generator* is a sequential system that produces, as output, a periodic pattern of elements which belongs to a certain set of output values. These patterns are characterized by the elements that are present in them and the length of the pattern, i.e. the period duration. Figure 6.20 shows three pattern examples with elements of the same set of output values. We see that in all cases that, after some time, a special sequence of output values is repeated. After the last pattern element is produced the system will begin, once more, with the first element of the pattern. A pattern generator can be realized by a finite state machine. For a periodic length n we make the next state function equal to the next state function of a modulo-n counter. Thus, we have n states. The output function is now a simple mapping of each state to the corresponding output value.

Figure 6.21 shows an example. We want to describe a pattern generator which sequentially generates the following output values 0, 5, 5, 9, 9, 0, ... etc. Thus, the pattern has a period of length 5. The next state function is equal to that of a modulo-5 counter. The rest of the behavioural description is simple. Afterwards we define the range of the output and the state set as type declarations. We can declare the necessary variables. Notice also that the pattern generator is controlled by an external event, which is defined as the variable Sync. The following output value is determined in the procedure SET_NEXT_OUTPUT.



PATTERN GENERATOR - EXAMPLE (continued)

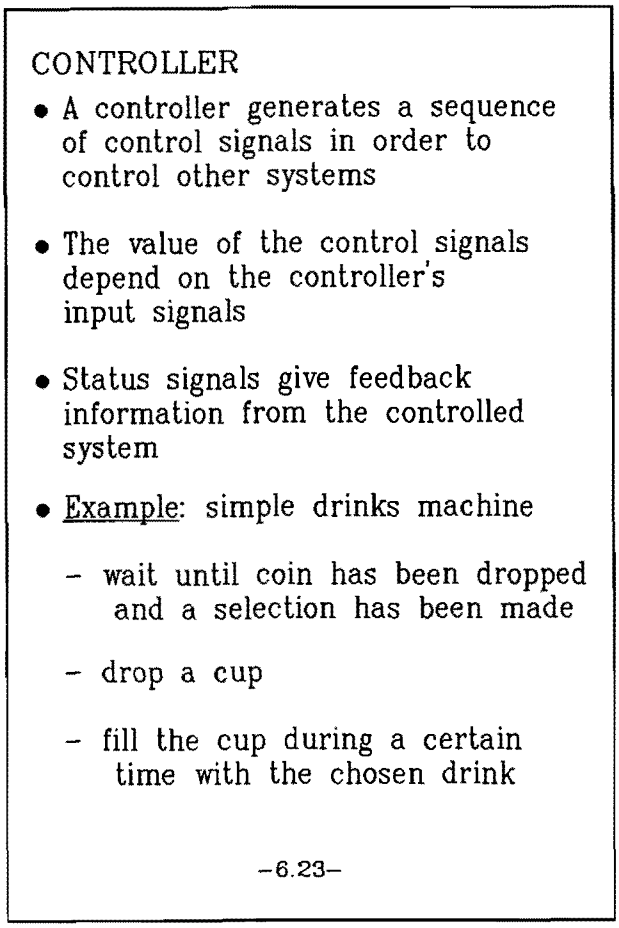
```
PROCEDURE Set_Next_Output;
  BEGIN
      (* determine NO:—>S *)
    CASE State OF
      0,1: Output := 5;
      2,3: Output := 9;
        4: Output := 0
    END
  END;
```

-6.22-

This procedure is described separately in figure 6.22. In the REPEAT FOREVER loop we see again the known behavioural description. First we wait for the external event. Then we set the following output value, and then go to the following state. Notice that the way in which we determine the following state is completely similar to the method applied in a modulo-n counter.

As we have said, figure 6.22 describes the procedure of determining the following output value. Notice that we still must specify the new output value from the current state. In state 0 and 1 the new value is 5, in state 2 and 3 the new value is 9, and in state 4 the new output value is 0. This behaviour is described in the shown CASE statement.

The controller forms yet another class of sequential systems. A controller is a sequential system (see figure 6.23) which generates a sequence of control signals by which other systems or subsystems can be controlled.

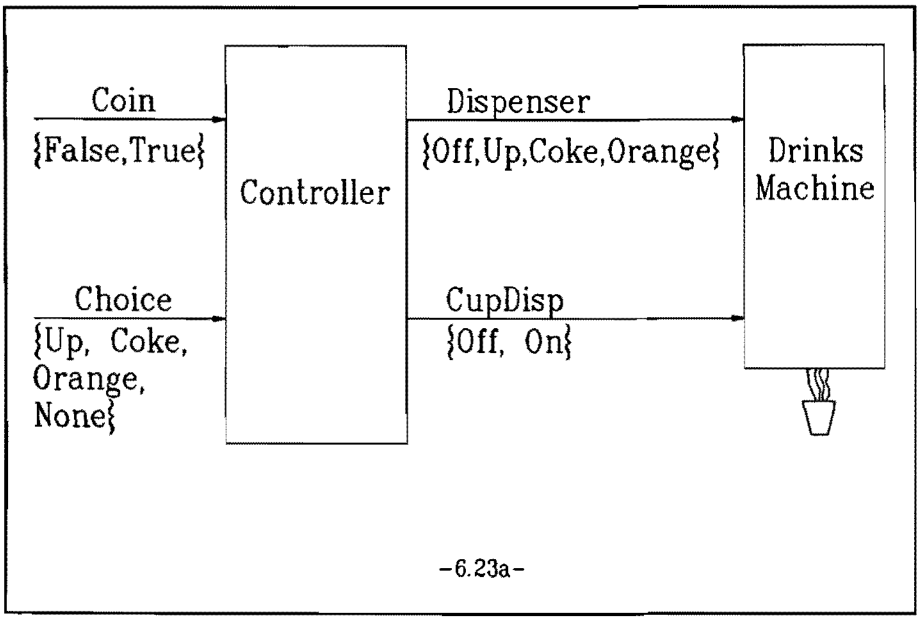


CONTROLLER

- A controller generates a sequence of control signals in order to control other systems
- The value of the control signals depend on the controller's input signals
- Status signals give feedback information from the controlled system
- Example: simple drinks machine
  - wait until coin has been dropped and a selection has been made
  - drop a cup
  - fill the cup during a certain time with the chosen drink

-6.23-

Till now it appears that the definition of a controller strongly corresponds to that of a pattern generator. The difference is that a controller reacts to values on its input. The generated control signals are in general depending on the input values. These input values are frequently called *status signals*, used by the control system to return its current state. For example, a motor's status could be halted or running; a container's status could be its amount of liquid; a kettle's status could be its momentary pressure etc. We could make use of a controller to, for example, determine a minimum liquid level in a container. In this course we shall, as an example, discuss a simple controller for a drinks machine. Figure 6.23 summarizes what this controller must do. First it waits until a coin in inserted and a choice is made. These are the two inputs of our controller. After that the controller will cause a cup to be dropped, and after a certain waiting time, it will fill it with the chosen drink. The controller is simple in the sense that we can go through all different problems that can occur, such as the unavailability of cups, the drinks running out or the improper functioning of specific items.


Coin
{False,True}
Choice
{Up, Coke,
Orange,
None}
Controller
Dispenser
{Off,Up,Coke,Orange}
CupDisp
{Off, On}
Drinks
Machine

-6.23a-

```
DRINKS MACHINE - BEHAVIOURAL DESCRIPTION (1)
SYSTEM DrinksMachine;
CASE State OF
     Wait : IF Coin AND (Choice<>None)
               THEN BEGIN
                 CupDisp:=On;
                 State:=DropCup;
                END;
  DropCup: BEGIN
              WAIT_FOR_EVENT(Timer);
              CupDisp:=Off;
              CASE Choice OF
                 Up : Dispenser:=Up
                Coke: Dispenser:=Coke;
              Orange: Dispenser:=Orange
              END; (*Choice*)
              State:=DispDrink
           END;
  DispDrink: BEGIN
              WAIT_FOR_EVENT(Timer);
              Dispenser:=Off;
              State:=Wait
           END
END; (*State*)
```

-6.24-

As shown in figure 6.24 the behavioural description is nevertheless somewhat complex. First we distinguish the three controller states. These are "Wait", the controller waits for the coin and the choice of drink type. Then we have the state "DropCup", where a cup is being dropped. Finally we have the state "DispDrink" where the cup is filled with the chosen drink.

These states are defined in the type STATE. Also, we have summarized the choices that can be made in the type CHOICES. Notice also that no choice is a possible value. The possible control signals for the drinking dispenser are summarized in the type DISPENSE. Finally, releasing a cup happens due to a signal of the type CUPREL. Using these types, we can declare a number of necessary variables. These are the variables State, Choice, CupDisp, and Dispenser. Notice that we make use of the boolean Coin and the variable timer of the type EVENT. The behavioural description begins with defining an initial state and its corresponding output values. After that the known REPEAT FOREVER construct, depending on the current state, determines what the new action must be and what the new state must become.

```
DRINKS MACHINE - BEHAVIOURAL DESCRIPTION (2)
SYSTEM DrinksMachine;
TYPE STATES=(Wait,DropCup,DispDrink);
     CHOICES=(Up,Coke,Orange,None);
     DISPENSE=(Off,Up,Coke,Orange);
     CUPREL=(Off,On);
VAR State     : STATES;
    Coin      : BOOLEAN; (*True,False*)
    Choice    : CHOICES;
    Dispenser: DISPENSE;
    CupDisp   : CUPREL;
    Timer     : EVENT;
 BEGIN
   State := Wait; (*BeginState*)
   Dispenser := Off; (*BeginConditions*)
   CupDisp := Off;
   REPEAT
     CASE State OF
         |
         |
         |
     END (*CASE*)
   FOREVER
 END.
```

-6.25-

We do this with the CASE statement described in figure 6.25.

In the state "Wait" we wait until the coin is inserted and the choice is made. Then we give the free cup signal and we go to the "Drop Cup" state. In this new state we wait for some time to allow the cup to become stable, after which we give the drink dispenser the correct information. At the same time we go to the state "DispDrink". Here we allow the selected drink to be poured for a fixed amount of time after which the dispenser is switched off and goes to the initial wait state. This completes the behavioural description of the controller of our drinks machine.

The given description leads to two remarks. We notice that the next state function and the output function are not described in separate procedures, but are both described in a CASE statement. Furthermore, we see that now the output signal "Dispenser" depends on "Choice", i.e. on an input signal. Thus, the value of Dispenser is not exclusively determined by the machine's state.

ASM

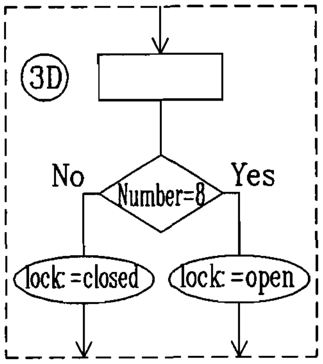
chart

-6.26-

ALGORITHMIC STATE MACHINE CHART

An ASM chart

- is a behavioural description of a synchronous system (FSM)
- uses 3 different symbols



state box

decision diamond

output box

- each state has a single entry
- a decision diamond performs
  - a single test on a single input

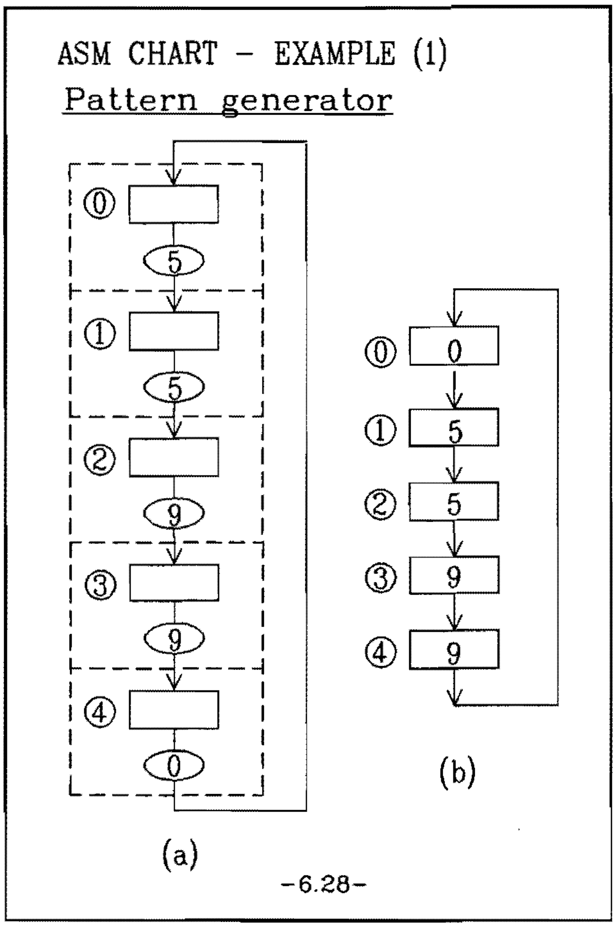
-6.27-

6.3 ASM chart

Next to the use of formal languages such as Pascal, there are other popular methods to describe the behaviour of finite state machines. These methods frequently have a less formal character than a language, but are a little closer to a possible realization. A method to be discussed in this course is the use of the *Algorithmic State Machine* (ASM) chart. This is a description method based on graphical symbols. A central concept in an ASM chart is the machine state. An assumption implicitly used in this method is that the machine can only go to a following state after a command of the global synchronization mechanism. We use an ASM chart to describe the behaviour of synchronous systems. Figure 6.27 shows that we can make use of three different symbols in the behaviour description of a finite state machine. The state of the machine is given by a rectangle, the *state box*. The name of the state, the state symbol, is shown to the left of the state box inside a circle. See state 3D in figure 6.27.

Under the state box it is possible to further specify the behaviour of the machine in this state. We do so by using one or more *decision diamonds.* These decision diamonds are used for testing input values. The requirements of these input values must be formulated in a way that can be answered with "yes" or "no". After these decision diamonds one or more *output boxes* are used. These output boxes are used to specify the *following* output values, i.e. for specifying the next output function NO.

The whole part surrounded by the dotted line in figure 6.27, describes the behaviour of a state machine for one state, in this case the behaviour of our electronic lock for state 3D. An important aspect of an ASM chart is that we can only enter the description of an arbitrary state in one way, that is in the top side of the state box. The behavioural description of a state can only be given via one of the output boxes. Furthermore, it is important that we have to use an individual decision diamond for each input we want to test in a state.



ASM CHART - EXAMPLE (1)

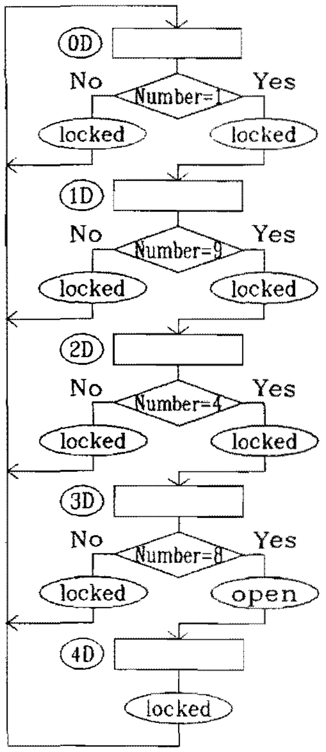
Pattern generator

-6.28-

We shall now consider a number of examples of behavioural description using an ASM chart. In figure 6.28 we find the behavioural description of our pattern generator. Here we have the behavioural description of each state surrounded by dotted lines. Notice that in our pattern generator there are no inputs to be tested, thus, the ASM chart has no decision diamonds. We notice also that each state has only one output box; accordingly we can only go to a single following state from each state. Then the output in each state can only have a single value.

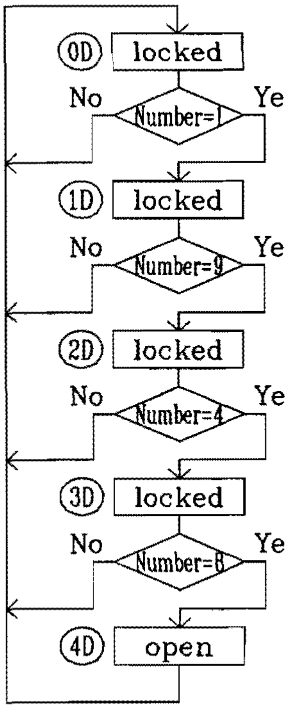
In such a situation we can introduce a simplification. We can write the output value in the state box of the following state, where we can dismiss the separate output boxes. The resulting ASM is shown in figure 6.28b. Notice that now the description of a state consists of only a state name (or state symbol), and the state box where we have written the current output value. Verify that it is still the previous state that determines the current output value. Later we shall discuss a realization form for which the interpretation is completely different.

ASM CHART - EXAMPLE (2)
Electronic digit lock



-6.29-

ASM CHART - EXAMPLE (2)
Electronic digit lock (continued)



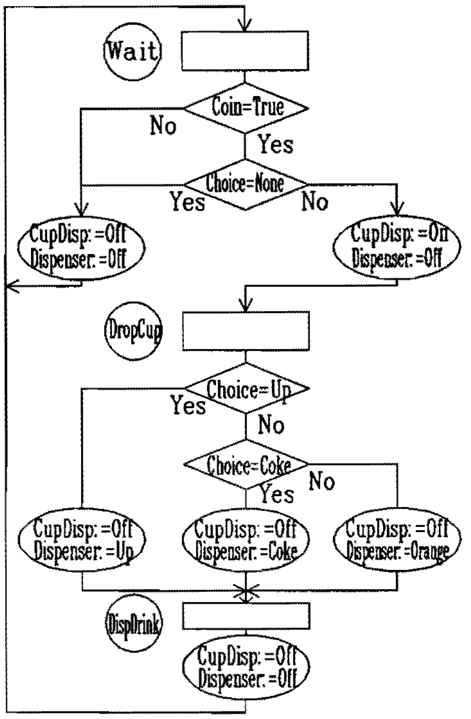
-6.30-

In figure 6.29 we find the ASM chart description of our electronic digit lock. Compare this description with the Pascal-based description of figures 6.16 and 6.17. From now on we shall dismiss the dotted lines by which we surround the description of one state. These dotted lines are officially not a part of the ASM description. Notice that in the Pascal-based description we also define the value sets of the inputs, states and outputs. However, in an ASM chart there are no good means to define these value sets.

We shall have to deduce them from the context. The behaviour of the finite state machine for our electronic digit lock, as described in figure 6.29, speaks for itself. Notice that we can still arrive at a single final state from the states 1D, 2D, 3D, and 4D. Consequently the output value remains the same in these states . Furthermore, we see that in state 0D the output value remains "locked". Also now the output can take only one possible value in each state. The ASM chart can be simplified again by writing this output value in the state box of the following state and omitting the output boxes. This is done in the ASM chart of figure 6.30. Check for yourself that this ASM chart corresponds with that of figure 6.29.

ASM CHART - EXAMPLE (3)

Drinks machine



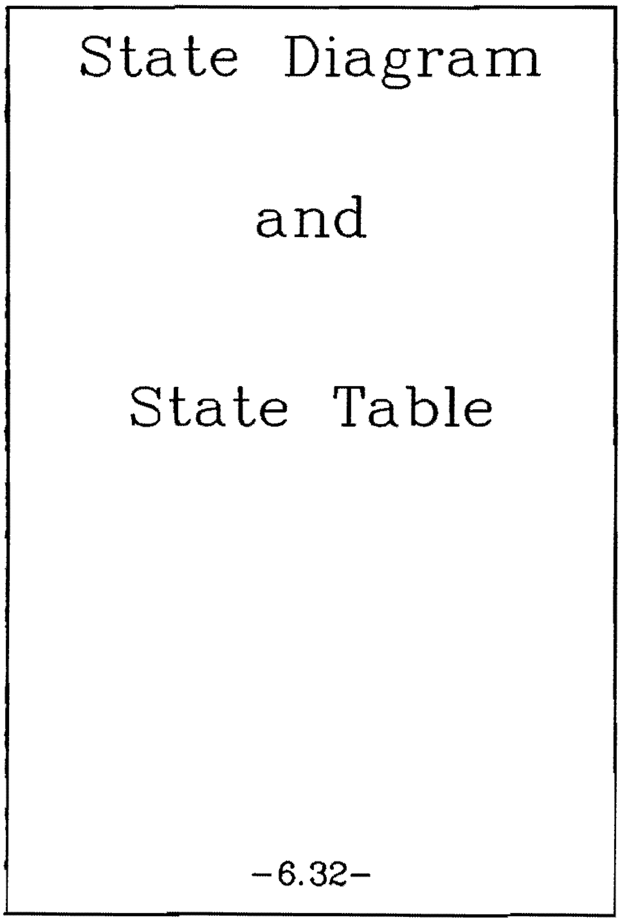
-6.31-

Finally, we have shown in figure 6.31 the ASM chart of our drinks machine. Notice that we need to test more than one input value in two states. Thus, we use more decision diamonds. Furthermore, the behavioural description completely corresponds to the previous behavioural descriptions given in the Pascal-based language (see figure 6.25). Check this for yourself. Because in the state "DispDrink" the value of the output depends on the choice made, we can not write the output value in the state box of the following state. A small simplification is possible as shown in the next problem.

Problem 6.3 (6.3)

The ASM chart of the drinks machine (figure 6.21) shows that in the states "Wait" and "DropCup" the outputs can take only one possible value. Draw, based on this, a simplified ASM chart.

We have now learned how to produce a behavioural description of a finite state machine using the ASM chart technique. With the ASM chart we can formally define the behaviour of our finite state machine. The strong side of the method is the uniform way in which a state's behaviour can be described. Decision diamonds create a clear decision structure. It is always clear in which path the state goes; what will be the following state and the following output value. The weak side of an ASM chart is the lack of means for specifying the value sets of input, states and output. Nevertheless an ASM chart is a valuable support tool. As we shall later observe, it has a strong resemblance to a realization method which we shall apply to finite state machines.

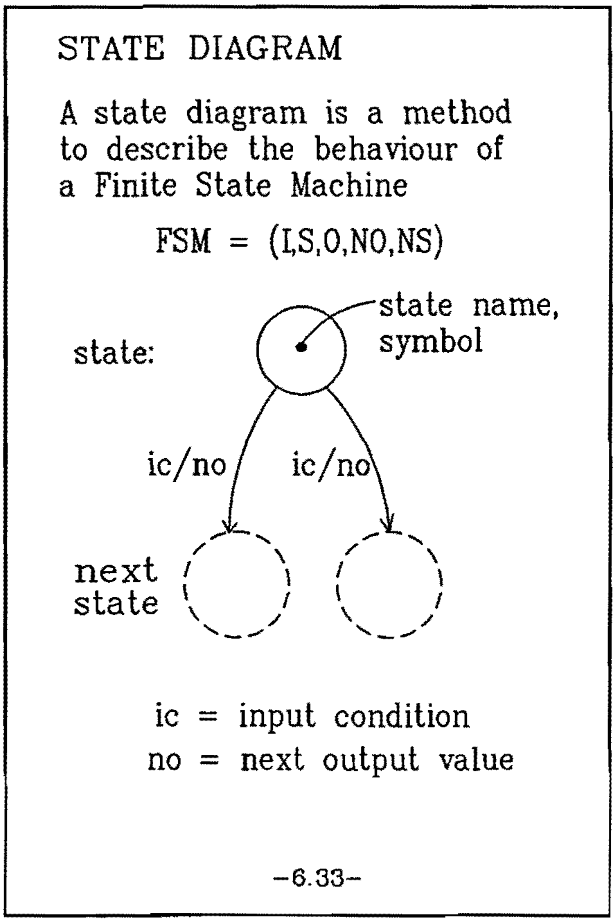


State Diagram

and

State Table

-6.32-



STATE DIAGRAM

A state diagram is a method to describe the behaviour of a Finite State Machine

FSM = (I,S,O,NO,NS)

ic = input condition

no = next output value

-6.33-

6.4 State diagram and state table

Another method for describing the behaviour of a finite state machine makes use of the *state diagram.* Also, in a state diagram the starting point is formed by the machine states.

A state is shown as a circle which is drawn around the state name or state symbol. We show that we can go from one state to the other using arrows between the two states. Both are shown in figure 6.33. Notice that we write the so-called *input condition* and next output values near the arrows.

STATE DIAGRAM

ic = input condition:

- conditions to be fullfilled by the input values in order to go to the indicated next state
- inputs not mentioned have no influence
- only one of the conditions at the arrows leaving a certain state, may be true

no = next output value:

- a next output should be specified for
  - each arrow and
  - each output

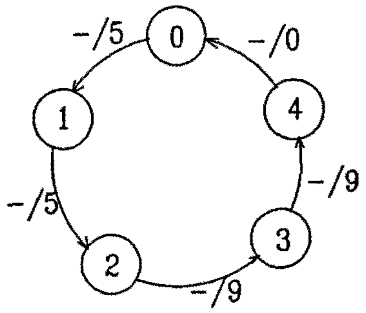
Several outgoing arrows may lead to the same next state.

−6.34−

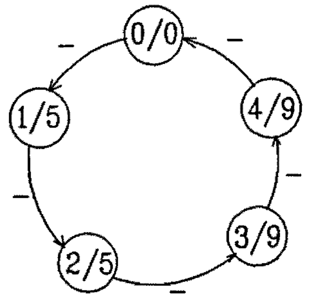
STATE DIAGRAM − EXAMPLE (1)

Pattern generator

- No input conditions



- Only one output value for each state



−6.35−

In figure 6.34 we go further into details. By an input condition we mean the requirements to be fulfilled by the input values in order to go to the indicated following state. Naturally, this state transition will only take place upon a command of a special synchronization signal.

Specifying all inputs at the input conditions often involves unnecessary writing work and leads to unclear conditions. Therefore, the convention should be followed that inputs have no influence on state transitions not referring to them.

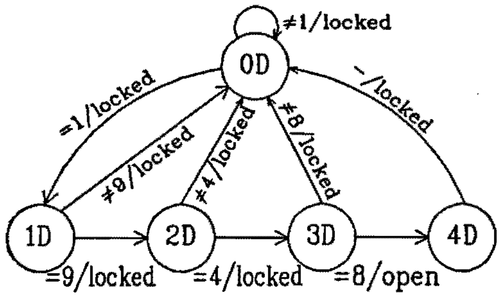
In a good behavioural specification it must be absolutely clear which is the following state. Applied to our input conditions this means the following: if, from a given state, several arrows go to following states, then the different input conditions corresponding to these transitions must not be true at the same time. More precisely: there may be at most one true condition among those leaving a state. After the input condition, separated by a "/" we place the next output value. Here we need to specify for all outputs (per output arrow) the following output value. The clarity of the specification plays an important role. There should not be any doubt about the value of the output in a following state.

Therefore we also agree that at each arrow only one following value may be specified for each output. However, we may specify different new output values; there may be several arrows to the same following state. We shall now illustrate the use of the state diagram through a number of examples.

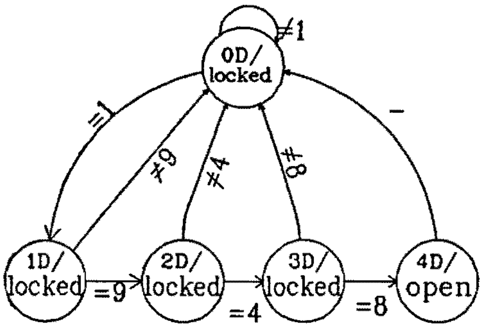
In figure 6.35 we have first drawn the state diagram for our pattern generator. It is easy to recognize that we always go through the different states, one after the other in the same sequence. Notice that the fact that there are no inputs to be tested (there are no input conditions) is specified by a horizontal dash. The output value follows the slash. This is always the output value of the pattern generator in the state to which the arrow is pointing. In state 1 and 2 the output is 5, in state 3 and 4 it is 9, and in state 0 it 0. As we noticed in the ASM chart, also here we have to deal with a situation where the machine in each state produces only one output value. In this situation we can further simplify the state diagram, by writing these output values in the following state circle, with the slash (/) separating it from the state symbol or the states name. This simplified state diagram Is also shown in figure 6.35.

STATE DIAGRAM - EXAMPLE (2)

Electronic digit lock



- only one output value in each state



-6.36-

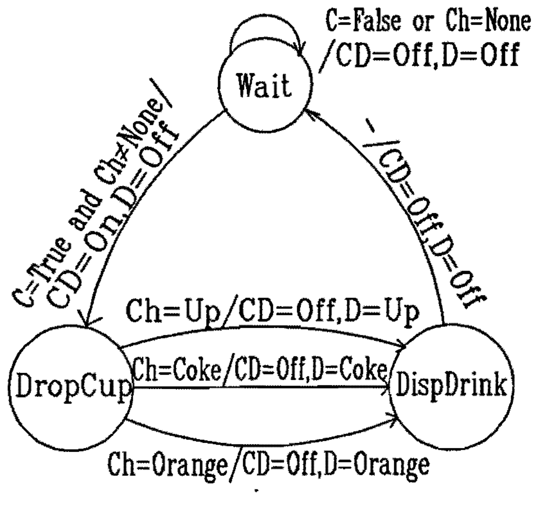
STATE DIAGRAM - EXAMPLE (3)

Drinks machine

- Abbreviations:

C=Coin ; Ch=Choice

CD=CupDisp ; D=Dispenser



-6.37-

In figure 6.36 we have drawn the state diagram of our electronic digit lock. Notice first that we can go back to state 0D from all states. Also from state 0D an arrow is drawn with its beginning and ending points on the circle of state 0D. Here we show that as long as we do not choose the digit 1 first, we remain in state 0D. Notice further that from each state, except state 4D, 2 arrows go towards two following states. Here we have to deal with the requirement that only one of the input conditions of these arrows may be true. The fact that this is fulfilled in figure 6.36 follows directly from the way we have written the input condition.

Because also now the output in each state can take only one value, we can simplify the state diagram. This is shown in figure 6.36. Also now we can interpret this as a specification of the current output value in the current state. Notice for example that the lock is open in state 4D, after four correct digits are input.

We finally see in figure 6.37 the state diagram of our drinks machine. To limit the length of the input conditions and specification of the next output values, we have applied a number of abbreviations. Notice that we now specify the values of the outputs CupDisp and Dispenser at each state transition. Give further attention to the input conditions that are shown at the outgoing arrows of the "Wait" state. Convince yourself that the condition "C=False or Ch=None" and the condition "C=True and "Ch≠None" can never be true at the same time. With the help of De Morgan's theorem we can convert one condition to the other. Notice finally that we have three arrows going to the state "DispDrink" to show the three different values of the output Dispenser.

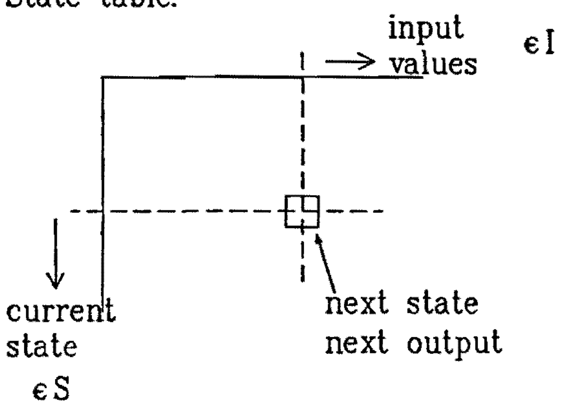
STATE TABLE

A state table is a method to describe the behaviour of a Finite State Machine

Behavioural description:

$$FSM = (I,S,O,NO,NS)$$
$$NS: S * I \rightarrow S$$
$$NO: S * I \rightarrow O$$

State table:



-6.38-

STATE TABLE - EXAMPLES (1)

Pattern generator

- No input values

current state	
0	1,5
1	2,5
2	3,9
3	4,9
4	0,0

next state,
next output

Electronic digit lock

current state	input values									
	0	1	2	3	4	5	6	7	8	9
0D	0D,l	1D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l
1D	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	2D,l
2D	0D,l	0D,l	0D,l	0D,l	3D,l	0D,l	0D,l	0D,l	0D,l	0D,l
3D	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	4D,o	0D,l
4D	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l	0D,l

o=open
l=locked

next state, next output

-6.39-

A following step in the specification of finite state machines - a step closer to realization - is the use of the state table. The ASM chart and the state diagrams are two graphical methods for the specification of behaviour. The state table is surely not. With a state table we concentrate on the next state and next output functions, NS and NO. For both functions it is valid that their domain is given by the cartesian product of the set of states and the set of input values of the finite state machine.

We create a two-dimensional structure, a matrix, where the current state selects a row and where the input values define the columns (see figure 6.38). At the intersections between the machine states and the input values we can specify the following state and the following output value. We have now a two-dimensional table, called a state table. In this state table we can directly fill in the next state function and the next output function.

In figure 6.39 we have first constructed the state table of our pattern generator. Because there are no inputs, our two-dimensional matrix structure is reduced to one column. In each row (per current state) we write what will be the next state and the next output. If we look at the ASM chart in figure 6.28 or the state diagram in figure 6.35 we see that we can simply derive the state table from one of both specifications. This is a useful practical approach.

Our electronic digit lock is a finite state machine where inputs play a role. The state table shown in figure 6.39 thus has several columns, one for each input value. In the resulting table we have shown, for each state, an input value, the next state, and the next output. Here we have indicated using the letter l that the lock is locked and with the letter o that the lock is open. From this example it appears clearly that working with input values is not always practical. We must now specify 50 new states, at the same time, in the previously given state diagram or ASM chart it was limited to 9 new states.

STATE TABLE - EXAMPLES (2)

Electronic digit lock (continued)

- use of input conditions

current state	=1	=9	=4	=8	≠1and≠9and≠4and≠8
0D	1D,1	0D,1	0D,1	0D,1	0D,1
1D	0D,1	2D,1	0D,1	0D,1	0D,1
2D	0D,1	0D,1	3D,1	0D,1	0D,1
3D	0D,1	0D,1	0D,1	4D,o	0D,1
4D	0D,1	0D,1	0D,1	0D,1	0D,1

next state, next output

- 2-column table

current state, Input	next state, Output
0D, ≠ 1	0D, locked
0D, = 1	1D, locked
1D, ≠ 9	0D, locked
1D, = 9	2D, locked
2D, ≠ 4	0D, locked
2D, = 4	3D, locked
3D, ≠ 8	0D, locked
3D, = 8	4D, open
4D,—	0D, locked

-6.40-

Here we can do something by using input conditions instead of input values. This is done in figure 6.40. We have defined the following input conditions for our electronic digit lock: "the digit=1", "the digit=9", "the digit=4", "the digit=8", or "the digit equals non of these values". We see by doing so that the number of following state and output values to be specified is decreased to 25. Nevertheless we have input conditions that may play no role in certain states. So in state 0D we only have to check if the input digit is 1 or not; the conditions =9, =4, and =8 play no role in this state. In such a situation it is better to leave the two-dimensional structure in favour of a table with current state and input conditions next to each other. In figure 6.40 this two-column table has been worked out for our electronic digit lock.

Notice that we now only need to specify the next 9 states and next outputs, corresponding to our state diagram or ASM chart. Notice that we might have several rows (in a table) for the same state, but with another input condition. It is again emphasized that for a correct specification of these conditions, only one may be true. Indeed, it must always be clear what is the following state and what is the corresponding output value.

Problem 6.4 (6.4)

Construct a state table for the drinks machine. Explore both possible forms of the state table.

The state table forms the connection between the behavioural specification of a finite state machine and its realization. In the following chapter we shall come back to this.

Standard Architecture

and

Canonical Form

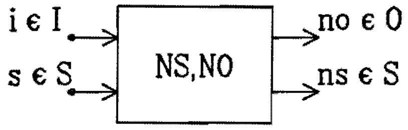
-6.41-

REALIZATION OF FINITE STATE MACHINE

- FSM specification

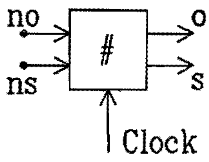
  FSM = (I,S,O,NO,NS)
  NS: $S * I \rightarrow S$
  NO: $S * I \rightarrow O$

- Realization of combinational functions



- Realization of memory (register) functions

  Record the next state and next output when receiving a command (clock)



-6.42-

6.5 Standard architecture; canonical form

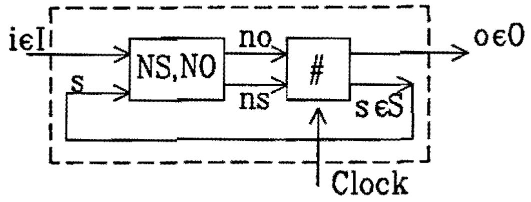
We have now discussed a number of methods for describing the behaviour of a synchronous finite state machine. For these types of machines we can now make our first step in the design process. In this paragraph we shall make a start with the question how to realize a finite state machine; in the next chapter this will be worked out further.

In principle we can realize finite state machines in several ways. There exists a useful general architecture, a generally applicable method for splitting into partial functions. We here refer to the *canonical* form (canon = rule, line of action). The names of Huffman, Moore and Mealy are connected to these canonical forms. This general architecture relates to our description of a finite state machine, where the machine is specified by a 5-tuple (I, S, O, NO, NS). Two important elements of the 5-tuple are the next state function NS and the next output function NO.

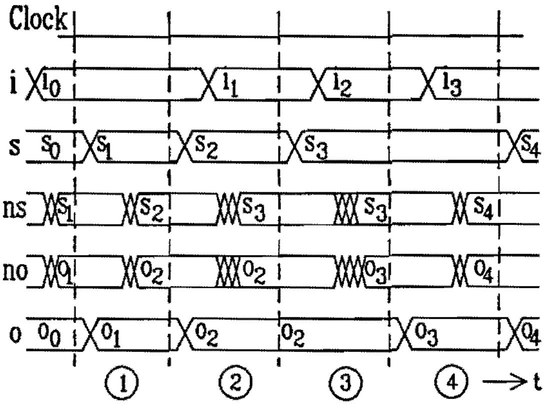
Previously, we have noticed that these are two combinational functions having the same domain. These combinational functions can be realized in one combinational block, as shown in figure 6.42. In previous chapters we have discussed how to realize these combinational functions. Notice that these functions determine the following state and the following output value, given the current state and momentary input value. For the realization of a finite state machine we still need a memory function. This memory function (when given a command) records the next state and the next output (which are determined by the combinational function) in order to replace the current state and the current output value. If we compare this required behaviour with the behavioural description of a register shown in figure 6.15, we notice that such a register is applicable here. As long as we do not know how to realize a register this remark is not important for now. At this moment it is sufficient to know that we can realize the necessary memory function using a register. We have now split the realization of a finite state machine into two partial realizations, namely the realization of two combinational functions and the realization of the memory function.

FSMs - STANDARD ARCHITECTURE

- Canonical form



- Timing diagram

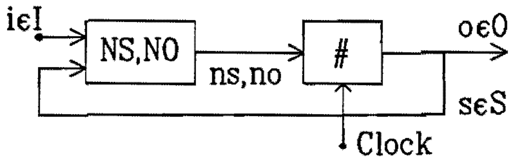


-6.43-

If we add those partial blocks together we get the standard architecture for finite state machines as shown in figure 6.43. We see that the combinational block determines the next output no and the next state ns from the momentary value of the input i and the current state s. On command of the input "Clock" the memory function takes over this next output and next state, and the current output and state are made equal to these new values. The timing diagram shown in figure 6.43 can clarify the behaviour further. Here we have the occurrence of the external event on the input "Clock" shown with so-called needle pulses. When interpreting the timing diagram we must consider that the execution of a combinational or memory function costs a certain amount of time. After the external event has taken place, it will take some time before the outputs s and o take new values. We see that clearly in time period 1, where s changes from s_0 to s_1 and o changes from o_0 to o_1. Subsequently the combinational block will get a new value on its input s, and after some time the next state s_2 and the next output o_2 will be determined. It is not given exactly at which time these value will be available. This uncertainty we indicate in the time sequence diagram by a number of crosses. In each case these new values have to be available before the following external event occurs. At that moment the new values are once more taken over by the memory function. We are now in time period 2. In this time period also the value of the input i changes; the next state s_3 and the next output o_2 are now determined by the momentary input i_1, and the current state s_2. Notice also that the value of the next output is equal to the value of the momentary output. Thus, the output value will not change and will remain equal to o_2. For a while the next output "no" takes no other values. Due to the synchronous nature of our finite state machine, this cannot cause any harm. The events in time period 3 and 4 speak for themselves. Notice also that the momentary state in time period 3 and 4 are equal; the state does not change.

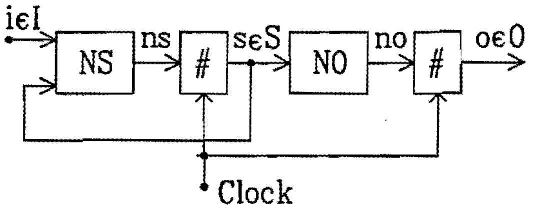
CANONICAL FORM - SUBTYPES (1)

Type 1 - Output and state set have common elements (possibly identical)



Type 2 - Next output is a function of current state only

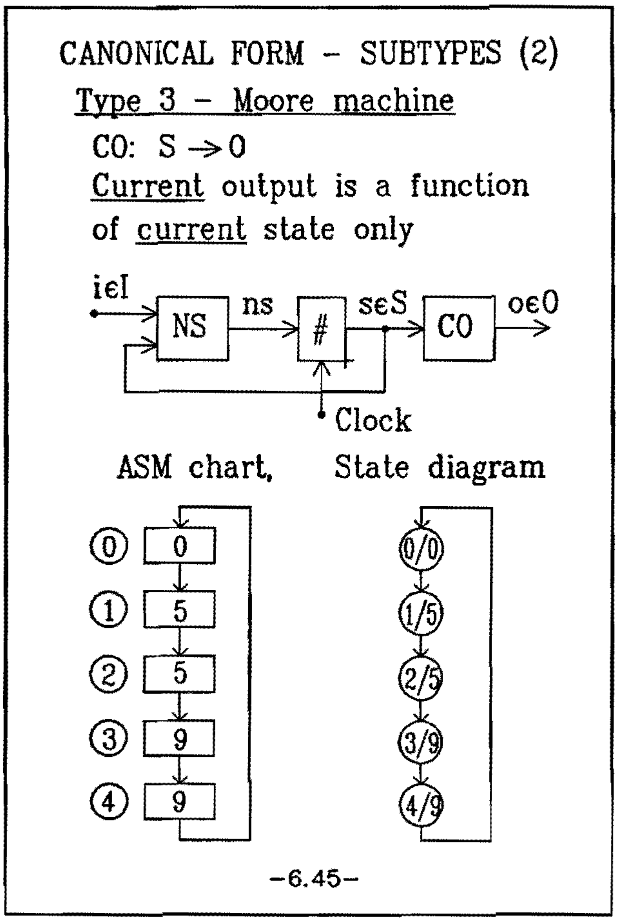
NS: S * I->S
NO: S ->0



-6.44-

This general architecture has a number of interesting variations. In figure 6.44 we show two of them. In the first place there are imaginable situations, where the set of states and the set of output values have common elements. In such a situation we can use the current state value also (partly) as an output value. We have illustrated this graphically by giving to the memory function only one output for the current output value and the current state value. This situation will arise frequently if the states and the output values are coded in binary tuples. We shall come back to this later.

A second variant we get if we consider that there are finite state machines for which the next output is exclusively determined by the current state and not by the momentary input. Our pattern generator and our electronic digit lock are examples of such finite state machines. In such a case it is better to separate the determination of the next state and the next output. For this purpose we use two separate combinational blocks. To also separate the memory function for the states and the output value, we get the architecture shown in the lower part of figure 6.44. A further variant can be obtained by omitting the memory function for the output values of this machine.



CANONICAL FORM - SUBTYPES (2)

Type 3 - Moore machine

CO: S → O

Current output is a function of current state only

-6.45-

We then get the finite state machine shown in figure 6.45. Notice that the output is not determined by the previous state but by the current state. In other words, the current state determines the current output value. Thus, we are considering the "current output" function CO. Such a finite state machine is called a Moore machine.

Notice that the previously simplified ASM chart or the state diagram, where the following output value was placed in the state box of the following state or in the circle of the following state, are also useful for the behavioural description of such a state machine. We only need to change the interpretation in: the current state determines the current output value. Thus, the state related to the corresponding state box or corresponding circle determines also the momentary output value. Furthermore, the behaviour is the same in general.

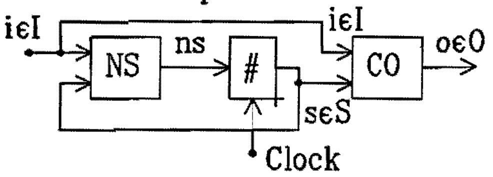
CANONICAL FORM - SUBTYPES (3)

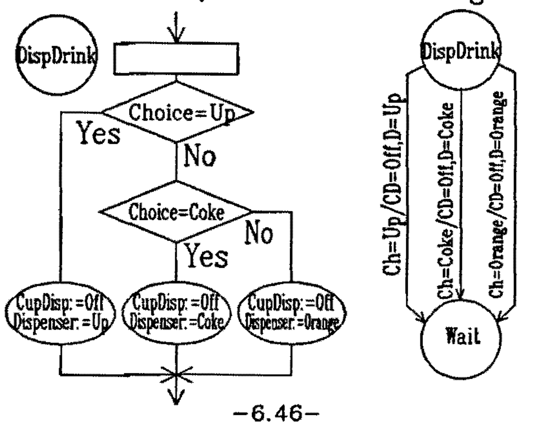
Type 4 - Mealy machine

CO: S * I → O

Current output is a function of

- current state
- current input values



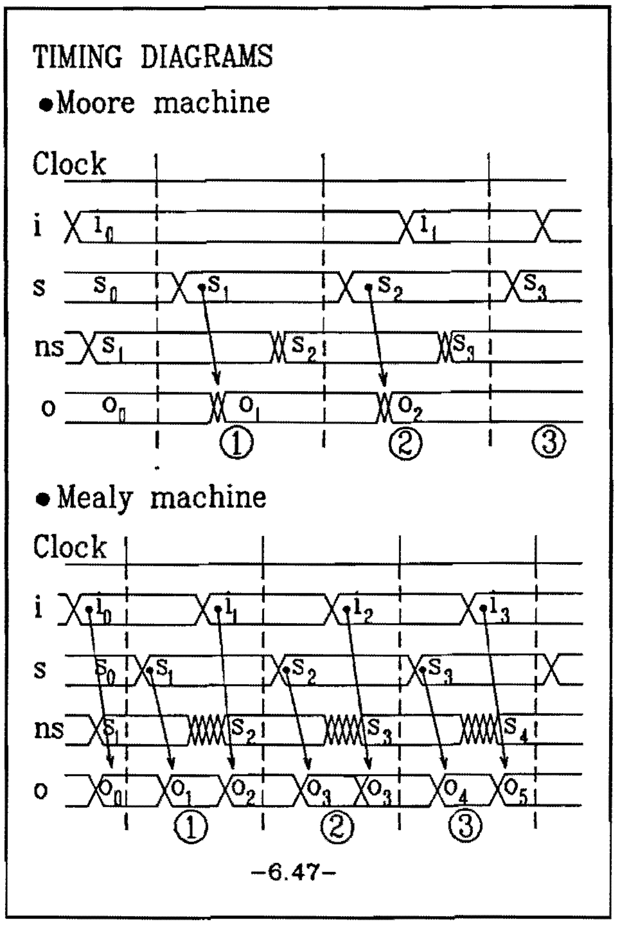

ASM chart, State diagram

DispDrink
Choice=Up
Yes
No
Choice=Coke
No
Yes
CupDisp:=Off Dispenser:=Up
CupDisp:=Off Dispenser:=Coke
CupDisp:=Off Dispenser:=Orange
DispDrink
Ch=Up/CD=Off,D=Up
Ch=Coke/CD=Off,D=Coke
Ch=Orange/CD=Off,D=Orange
Wait

-6.46-

The input of the current output function CO leads to a fourth variant on the general architecture, the Mealy machine. See figure 6.46. In this fourth variant the current output value is based on the current state and the momentary input value. The architecture of such a machine is shown in figure 6.46. Notice that we obtain this architecture by omitting the memory function for the Next Output function of our general architecture (in fig. 6.43). The behaviour changes now clearly. Indeed, a change of an input value now directly causes a change in output values. This change is not synchronized any more by an external event. We say that the changes of the outputs are *asynchronous*. A Mealy machine can very quickly react to changes in an input value. These asynchronous changes are at the same time sources of problems and errors, certainly in a system where several Mealy machines work together. Another problem occurs during the behavioural description of a Mealy machine.

It appears that our ASM chart or our state diagram cannot be used, unless we modify them. In figure 6.46 we have shown what needs to be changed. Because in a Mealy machine the current value of outputs is determined by the current state and the momentary value on the inputs, we must test these input values in the current state and not in the previous state, as we did earlier. In figure 6.46 we have shown a part of the ASM chart of our drinks machine for a Mealy machine. Comparing this part with the ASM chart shown in figure 6.31 we see that the decision diamonds for making the drinks choice, have moved from the state DropCup (DC) to the state DispDrink (DD). This corresponds completely with the fact that both the current state and the current input values determine the current output values. We have to perform the same change to the state diagram. Also this is shown in figure 6.46. Furthermore, we notice that the output boxes now contain the current output values, which are valid in the current state.


TIMING DIAGRAMS

•Moore machine

•Mealy machine

-6.47-

Figure 6.47 finally shows one timing diagram for the Moore machine and one for the Mealy machine.

Moore machine

In the timing diagram of the Moore machine it is shown with arrows that the current state s determines the current output o. We see first, in time period 1, that the current state changes from s_0 to s_1. Accordingly the output value will change after a short time from o_0 to o_1. At the same time the next state s_2 is determined. This is taken over by the memory function at the beginning of time period 2. A short time afterwards the new output value o_2 is determined. Notice that the change of the input values from i_0 to i_1 in time period 2 has no effect on the determination of the output values.

This input value only plays a role during the determination of the output value.

Mealy machine

In the timing diagram of the Mealy machine we clearly see (with the help of arrows) that a change of input values can have a direct effect on the momentary output value. In time period 1 we first see that the current state change from s_0 to s_1 causes the output value to change to o_1. A short time after this (in the same time period) the input value changes to i_1. Due to this the value of the output changes again, now to o_2. We see that in a Mealy machine the output may take several different values within one time period. The other time periods speak for themselves.

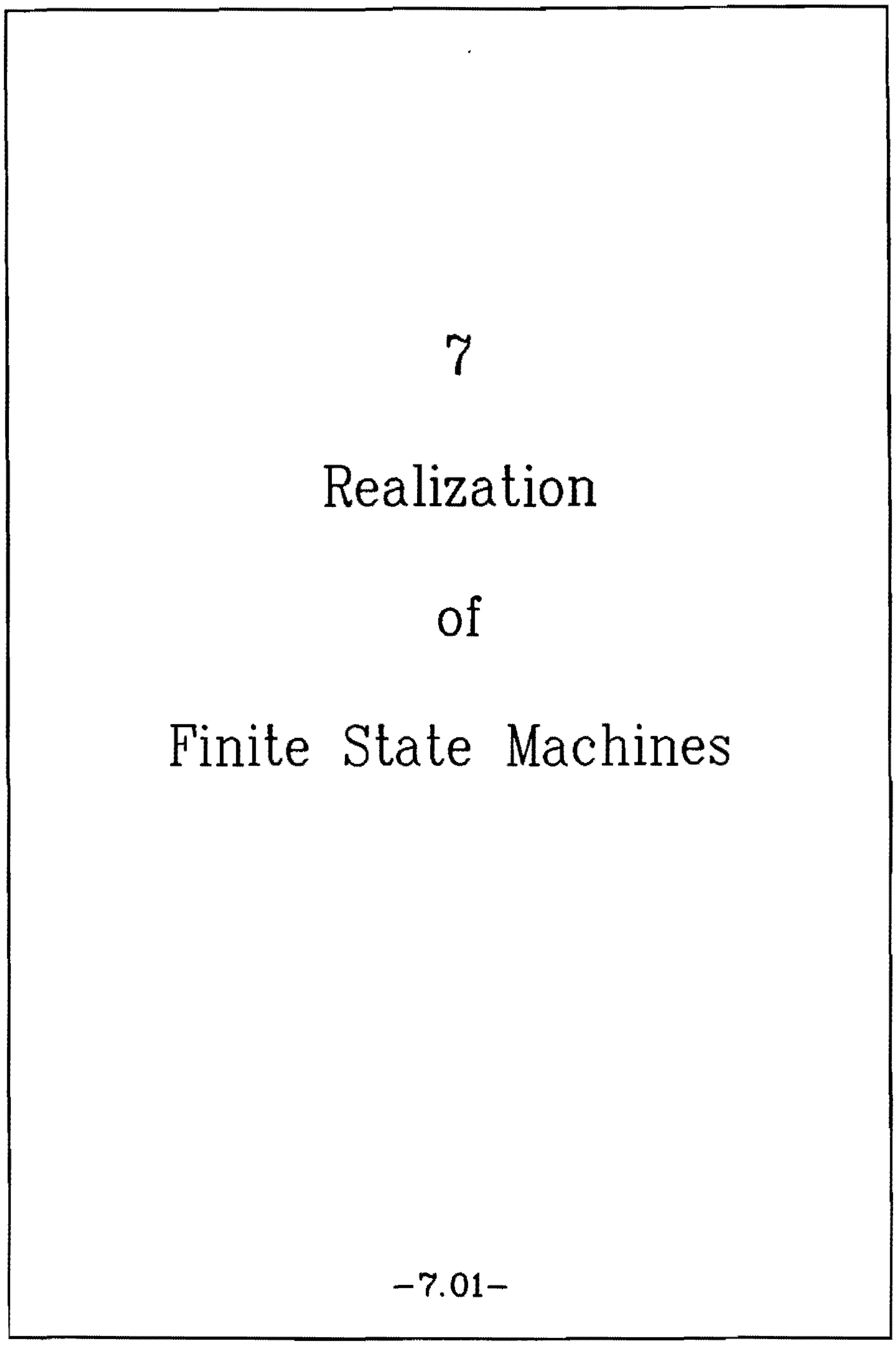
6.6 Summary

In this chapter we got acquainted with the characteristics of sequential systems. These systems are distinguished from combinational systems by their memory function. The response of sequential systems is not only depending on current input values, but also on previous ones. The system remembers what happened in the past. A special category of sequential systems is formed by the finite state automata or finite state machines. A special aspect of these machine is that the memory function is expressed in the machine state. There is a finite set of values, states, that represents the memory. The past is now accounted for in the momentary machine state. Next we have to deal with a finite set of input and output values for each digital system. A finite state machine is described by a 5-tuple, consisting of the elements: input set I, state set S, output set O and the functions for determining the next state NS, and the next output, NO. We have limited ourselves in this course to synchronous finite state machines, i.e. machines that go to the following state and set the following output value under the influence of an external event. We have discussed some methods for the behavioural description of such systems. For synchronization purposes we had to extend our Pascal-based language with two standard procedures WAIT_FOR_INPUT and WAIT_FOR_EVENT.

Next we introduced the standard type EVENT and the construct REPEAT - FOREVER. As exponents of finite state machines we have seen the modulo-n counter, the pattern generator and the controller. We have given these systems a Pascal-based behavioural description. Next we have learned about the ASM chart and the state diagram as two alternative methods for the graphical definition of system behaviour.

When working towards a realization, we shall translate the Pascal-based behavioural description to an ASM chart or a state diagram. Both methods are used in literature. After we have described our behaviour in a state diagram or an ASM chart, we can compose a state table. Such a table clearly determines the behaviour of the next state and output functions, and is a next step towards the realization of a finite state machine.

Finally we got to know a standard architecture or canonical form for the finite state machine. This standard architecture starts by separating the combinational logic block (where the next state and the next output functions are determined) and the memory function block. We have discussed a few variants of this standard architecture, including Moore and Mealy machines. These machines lack a memory function behind the next output stage. Here no next output, but rather a current output function is incorporated: the current state determines the current output value. In Mealy machines the momentary input value is included. Because the mentioned memory functions are lacking, these machines are, in general, cheaper to realize. The Mealy machine has the advantage that it can react quicker to input changes. A disadvantage of both machines is that outputs can change their values in short intervals; they do not change synchronously. This makes the design in general more difficult. Honesty compels us to say that till now the cost considerations have played a decisive role and, thus, one often chooses for Moore or Mealy machines. However, the costs of the memory function decreases with the continuously growing integration density, so maybe in future the more general canonical form will be chosen.



7

Realization

of

Finite State Machines

-7.01-

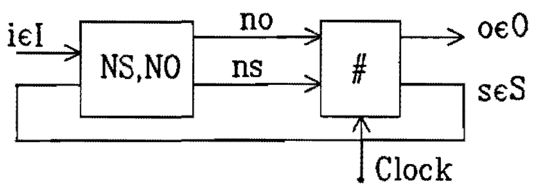
7 Realization of finite state machines

In previous chapters we discussed methods for describing the behaviour of finite state machines, that is, methods for their specification. At the same time we have considered the general structure, i.e. the architecture of finite state machines.

In this chapter we are approaching the second question in our design process: "How do I realize a finite state machine?". An important subitem is the realization of the memory function. What remains are two combinational functions, and we know how they can be realized.

FINITE STATE MACHINES - SUMMARY

- Definition
  FSM = (I,S,O,NO,NS)
- Canonical form



- Realization as binary system
  coding of I, S, O
- Realization of
  - memory;
    recording of bits
  - combinational functions:
    NS : I * S ⟶S
    NO : I * S ⟶O

-7.02-

Figure 7.02 shows in a nutshell the realization of FSMs. We know that a finite state machine can be described by a 5-tuple (I, S, O, NO, NS). In the previous chapter we have introduced the canonical form as a general architecture. The finite state machine is divided into two blocks, a block for the combinational functions NS and NO, and a separate block for the memory functions. We must consider that the realization will finally take place as a binary system. Accordingly our value sets I, S, and O must be coded. The realization of a finite state machine is thus carried out by first realizing the memory function, where the bits of the binary coded momentary state and momentary output are saved, and secondly by realizing the combinational binary functions NS and NO.

Realization of Memory

The Flip-Flop

-7.03-

7.1 Realization of memory; the flip-flop

We shall first concentrate on the realization of the memory function. Later the coding problem and some realizations of frequently used finite state machines will be discussed.

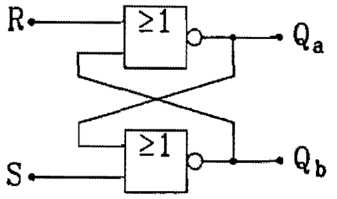
There are many known electrical and non-electrical methods to realize the memory function. As an example, one electrical method applies the charge saved in a capacitor. In this course, however, we shall specifically study a realization of memory functions generally applicable in finite state machines. Here the memory function is usually realized by a *flip-flop*. A flip-flop is a circuit with two stable states. The circuit always assumes one of these states. When externally affected the flip-flop may go from one of its stable states to the other.

A good mechanical analogy to the flip-flop is the light switch. It has two stable positions. By pressing on the right place the switch is tilted and goes over to the other stable position.

FLIP-FLOP REALIZATION

Memory function

- bi-stable circuit = flip-flop
  = circuit with 2 stable states



- Let S=R=0; Q_a=0, Q_b=1
  is a stable states
- Make S=1, R=0
  causes Q_b=0, then Q_a=1
  new stable state
- Make S=0, R=0
  Q_a=1, Q_b=0,
  remains in stable state
- Make S=0, R=1
  causes Q_a=0, then Q_b=1
  back to first stable state

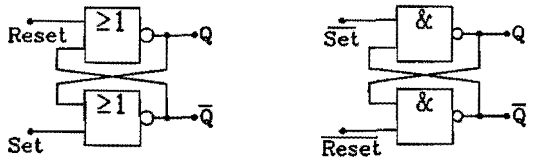
-7.04-

Figure 7.04 shows a possible realization of a flip-flop using two nor-gates. Let us analyze this circuit. First we assume that the inputs S and R are equal to zero, and that the output Q_a=0 and Q_b=1. One of the inputs of the upper nor-gate is now equal to 1 if the output remains 0. For the lower nor-gate both inputs are 0 when the output Q_b remains equal to 1. This situation does not change: the flip-flop is in a stable state. If we from this situation make S equal to 1 then one input of the lower nor-gate will be equal to 1; consequently the output Q_b will become equal to 0. This causes both inputs of the upper nor-gate to become equal to 0, where Q_a is equal to 1. After that nothing changes; the flip-flop is now in its second stable state.

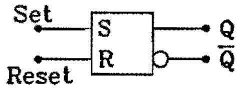
If we now make input S equal to 0, one input of the lower nor-gate will still remain equal to 1, and nothing is changed. The flip-flop remains in its second stable state. If we subsequently make input R equal to 1, one of the inputs of the upper nor-gate becomes equal to 1, and output Q_a becomes equal to 0. By doing so both inputs of the lower nor-gate become equal to 0, and output Q_b becomes 1. The flip-flop returns to its first stable state. Summarizing, we see that the flip-flop arrives at one of its stable states by making the input S equal to 1 for some time. We are considering the *set state*: the flip-flop is "set". The S input is called the *set input.* We can simply go further and state that, once the flip-flop is in its set state, making the set input 1 has no influence anymore. The flip-flop is set and remains so. By making the R input (the reset input) equal to 1 the flip-flop goes to its other state. We shall call it the *reset state.* We say that the flip-flop is "reset". Such a flip-flop is called a set-reset flip-flop or also a set-reset latch.

SET/RESET FLIP-FLOP (LATCH)

- with NOR gates
- with NAND gates



- Symbol



- Truth table

Set	Reset	Q_{t+1}	$\overline{Q}_{t+1}$	
0	0	Q_t	$\overline{Q}_t$	state not changed
0	1	0	1	reset state
1	0	1	0	set state
1	1	?	?	undefined

- Characteristic equation

$$Q_{t+1}=S+\overline{R}\cdot Q_t \text{ with } S\cdot R=0$$

-7.05-

In figure 7.05 we have shown that such a set-reset flip-flop can not only be realized with nor-gates, but also with nand-gates. In this case, we have to deal with not-set and not-reset inputs. In figure 7.05 we have shown the symbol for the set-reset flip-flop and the corresponding truth table. Notice that in the truth table we indicate the output value *after* the corresponding input condition is fulfilled as Q_{t+1}, and the output value *before* this input condition as Q_t. From this truth table we see that the state of the flip-flop remains unchanged if the set and reset inputs are both equal to 0. The reset state is entered if only the reset input is made equal to 1. Similarly we see that the set state is entered if only the set input is made equal to 1. One situation we did not discuss yet: what happens when the set and reset inputs are simultaneously equal to 1? Formally, this situation is not defined for the set-reset flip-flop. The result is depending on the realization of the flip-flop. In general we must avoid this situation when using a set-reset flip-flop.

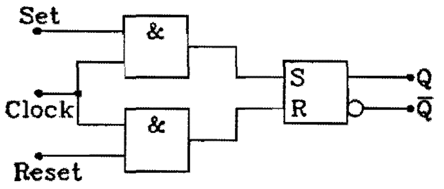
Problem 7.1 (7.1)

Analyze for the flip-flop realized with nor-gates the situation $S=R=1$, and show to which state the flip-flop returns if one of the input signals goes back to 0. What will happen is both signals return to 0 at the same time?

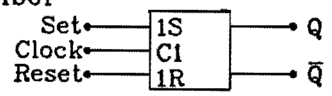
Next to the truth table we can also define the behaviour of a flip-flop in an algebraic way. This is done with the so-called *characteristic equation.* Figure 7.05 finally shows these characteristic equations for the set-reset flip-flop.

We have now got to know the flip-flop as a basic circuit for the realization of memory functions. Actually, for applications in our synchronous finite state machines, we miss one important element: the synchronization of the command input. In the case of flip-flops we consider the *clock input.*

SET-RESET LATCH WITH CLOCK



- Symbol



- Function table

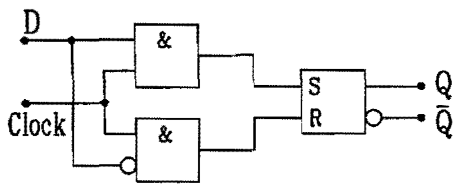
Clock	Set	Reset	Q_{t+1}	
0	x	x	Q_t	frozen state
1	0	0	Q_t	unchanged state
1	0	1	0	reset state
1	1	0	1	set state
1	1	1	?	undefined

- Characteristic equation

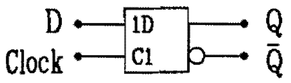
$$Q_{t+1} = C\cdot S + \bar{R}\cdot Q_t + \bar{C}\cdot Q_t \text{ with } S\cdot R=0$$

-7.06-

D-LATCH WITH CLOCK



- Symbol



- Function table

Clock	D	Q_{t+1}
0	x	Q_t
1	0	0
1	1	1

- Characteristic equation

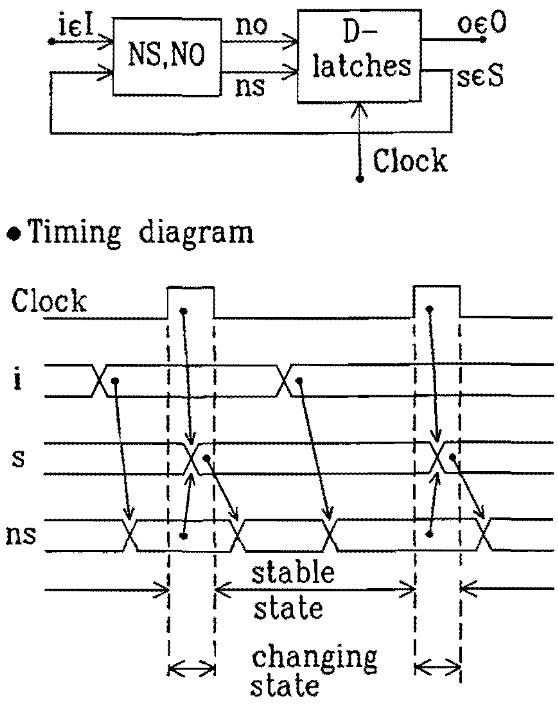
$$Q_{t+1} = C\cdot D + \bar{C}\cdot Q_t$$

-7.07-

Figure 7.06 shows how we can expand our set-reset latch with a clock input. For this purpose we "and" our set and reset inputs with the clock input. The effect is very simple, as shown in the function table of figure 7.06. As long as the clock input is 0 the values on the set and reset input play no role. The flip-flop keeps its state, we say that the state is *frozen*. Only when the clock becomes 1 (external command) the set and reset inputs get their previously discussed functions. This value dependency on the clock input is shown in the symbol of the set-reset latch. With a C at the clock input we show the so-called *command dependency*. The functionality of the other inputs depends on the value of this input. The other inputs have the same number; in this case 1 as the command input. The inputs 1S and 1R are depending on the value on the input C1.

In addition to the set-reset latch there are also other flip-flop types being of practical importance. One of them is shown in figure 7.07: the D-latch with clock input. This type is derived from the clocked set-reset latch. Let us consider the function table as a means of discussing its behaviour. We see that if the clock input is 0, the D input has no influence on the state of the flip-flop. If the clock input is 1, then a 0 on the D input will put the flip-flop in the reset state, causing the output Q_{t+1} to become 0. If in this situation the input D is made 1, the output Q_{t+1} will also become 1. We can also say that as long as the clock input is 1 the output will follow the D input. The behaviour is clear from this characteristic equation. Notice that the symbol again shows the command dependency of the D input on the value of the clock input.

D-LATCH - APPLICATION

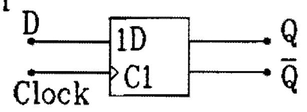


-7.08-

In figure 7.08 we see a number of D-latches used in a synchronous finite state machine. Clearly we recognize the canonical form where D-latches are used for the memory function. We shall explain their behaviour with the help of timing diagram. We begin with a 0 clock value. In this period, the value of the following state ns is determined from the current state s and the (eventually changed) input i. Actually, because the states of the D-latches are frozen, the value of s will not change yet. The new state is only assumed at the moment the clock input goes from 0 to 1. A short time later the outputs of the D-latches will change. This is expressed in a new value of the current state s. In the timing diagram we see that, a short time later, a new ns is formed as a result of the changes of s. This next state information may not yet be immediately taken over by the D-latches. Indeed, we assume only one state change per external command, per clock tick. Accordingly the clock input must be made 0 again in time. This is also shown in the timing diagram. We see that the states of the D-latches may only be changed during a short time. The pulses on the clock input are equal to 1 for a relatively short time. Similar pulses are difficult to generate and distribute reliably over a large system. In addition to that, the gates' delay time (and similar effects) play a relatively large role. These matters are indicating that D-latches are not really well suited for this application. Instead we should prefer flip-flops being input-sensitive (allowing their state to be changed) only for a very short time. An example is the edge-sensitive flip-flop.

EDGE-TRIGGERED D FLIP-FLOP

- Symbol

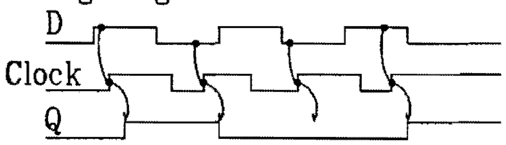


- Function table

D	Q_{t+1}
0	0
1	1

Q_{t+1} is the output value after a 0→1 clock edge

- Timing diagram



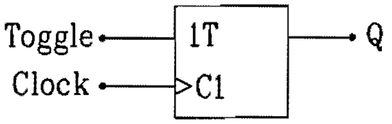
- Characteristic equation

$$Q_{t+1} = D$$

-7.09-

TOGGLE FLIP-FLOP

- Symbol



- Function table

Toggle	Q_{t+1}
0	Q_t
1	$\overline{Q_t}$

- Characteristic equation

$$Q_{t+1} = Q_t \oplus T$$

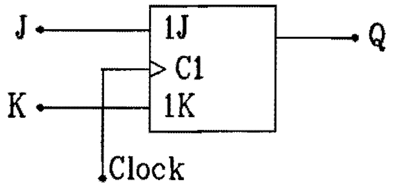
-7.10-

In figure 7.09 we have shown the edge-sensitive D flip-flop. Such a flip-flop can only change its state at the moment the clock input goes from 0 to 1. This behaviour is indicated by the triangle near the clock input. Also we notice this behaviour in the function table because the value of the clock input is not mentioned there anymore. A 0 to 1 transition is important at the clock input.
With Q_{t+1} we give the value of the output after such a transition at the clock input. We see that if the D input was equal to 0 for this transition, the new output value also becomes equal to 0. Also if the value of the D input was equal to 1 for this transition, then the new output value also becomes equal to 1. Again the output follows the D input but now with a certain delay caused by the clock input.
In the timing diagram we have shown also that the D input may change its value, even when the clock is 1, as long as the D input is stable just before and during the clock's transition from 0 to 1. We call this transition the *active transition.* The edge-sensitive flip-flop can not simply be realized as an expansion of the set-reset latch. The method of realizing this flip-flop is beyond the scope of this course.

Next to the edge-sensitive D flip-flop there are also other types. We present in figure 7.10 the T (toggle) flip-flop. The toggle flip-flop symbol shows that we are dealing with an edge-sensitive flip-flop. The function table clearly shows the behaviour of this flip-flop. If the value on the toggle input is equal to 0 then the state of the flip flop will not change; that the flip-flop is in *hold mode.* If the value of the toggle input is equal to 1 then after each active clock edge the output value and the state of the flip-flop will change. The flip-flop continuously goes, on command of the clock input, from one of its states to the other. Notice also that we can show this behaviour in the characteristic equation with the aid of exclusive-or function.

JK FLIP-FLOP

- Symbol



- Function table

J	K	Q_{t+1}	
0	0	Q_t	(hold)
0	1	0	(reset)
1	0	1	(set)
1	1	$\bar{Q}_t$	(toggle)

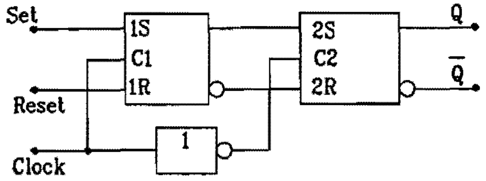
- Characteristic equation

$$Q_{t+1} = J\bar{Q}_t + \bar{K}Q_t$$

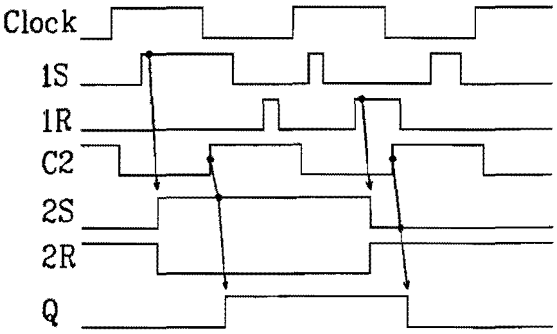
-7.11-

MASTER-SLAVE FLIP-FLOP

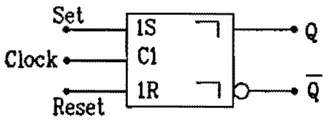
- Schematic diagram



- Timing diagram



- Symbol



-7.12-

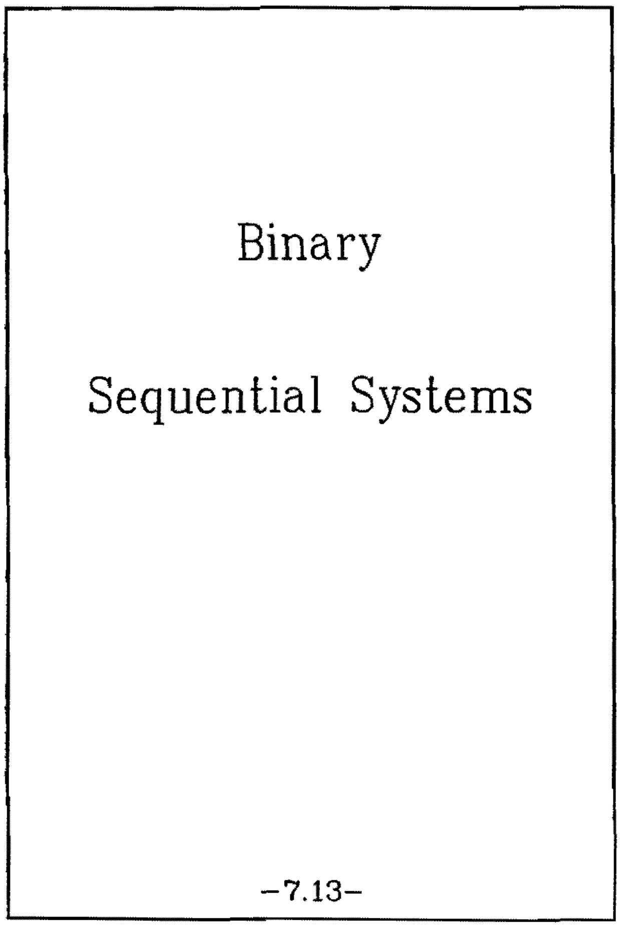
In figure 7.11 we have shown another type of edge-sensitive flip-flop, the JK flip-flop. The behaviour of this flip-flop is shown in the function table. Notice that if the J and K inputs are not both equal to 1, the behaviour is similar to a set-reset flip-flop. A 1 on the J input puts the flip-flop in the set state after the active clock edge; a 1 on the K input puts the flip-flop in the reset state after the active clock edge. If both J and K are equal to 0, the state of the flip-flop does not change.

Now the situation J=K=1 is defined: the flip-flop goes to its toggle mode, i.e. after each active clock edge the state is changed.

Remark:

We have discussed a number of flip-flop types reacting on a change of the clock from 0 to 1. There are also flip-flops reacting on a transition of the clock from 1 to 0. Such a flip-flop has the *negative edge* as the active edge. In the flip-flop symbol we show this by means of an inverter ball in front of the clock input.

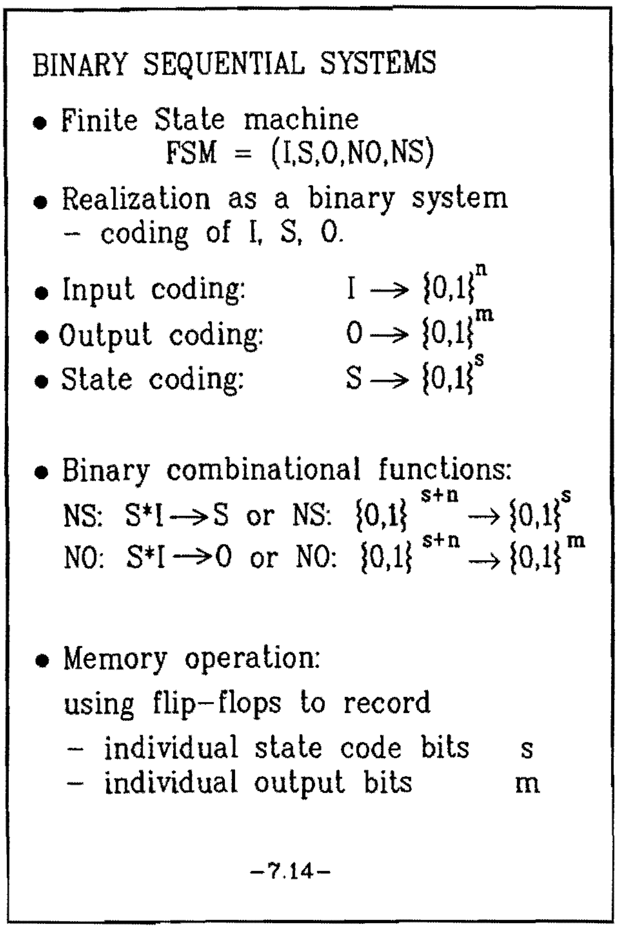
Edge-sensitive flip-flops are only sensitive to the values of its inputs for a short period. Another method to achieve a limited-period sensitivity is applied in the master-slave flip-flop shown in figure 7.12. We see that a master-slave set-reset flip-flop is built from two clocked set-reset latches, which are connected in cascade. The second latch gets the inverted value of the clock. If one flip-flop can change its state (it is *enabled*), the other flip-flop will freeze its state. While the right flip-flop freezes its state and the outputs remain constant, the first flip-flop follows the values of the set and reset inputs. When the clock input goes to 0, the left flip-flop will freeze its state to produce stable set and reset signals to the right flip-flop, which can now change state. The set and reset inputs have no influence anymore on this new state, because the left flip-flop has frozen its state. The total effect is that now the outputs of the master-slave flip-flop change at the moment the clock becomes 0. In this sense the master-slave flip-flop looks similar to the negative-edge sensitive flip-flop. However, the overall characteristics are still different, so the master-slave property is indicated by a separate symbol: a ¬ near the outputs. In addition to the master-slave set-reset flip-flop, the master-slave JK flip-flop is also a frequently used type.



Binary

Sequential Systems

-7.13-



BINARY SEQUENTIAL SYSTEMS

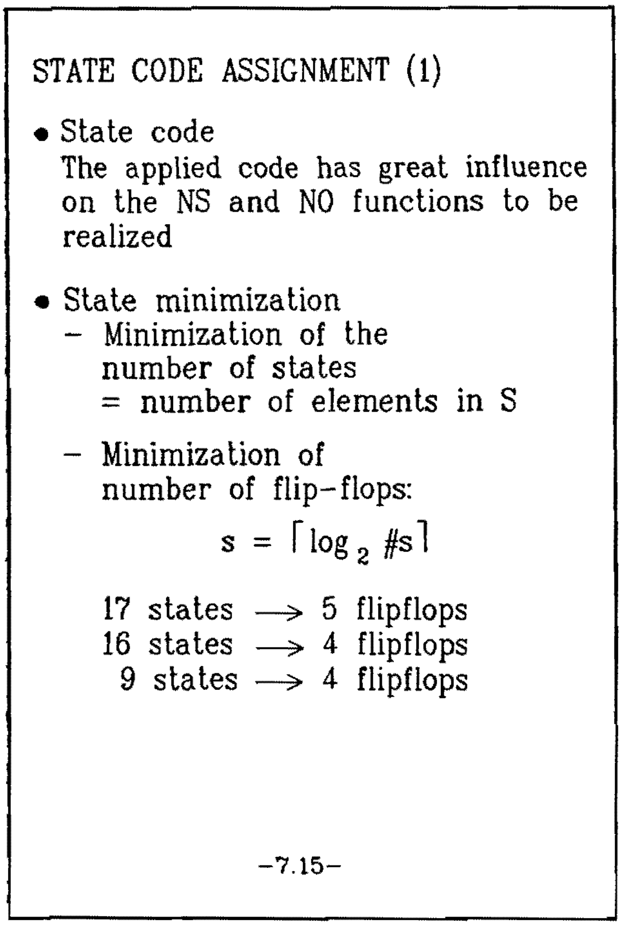
- Finite State machine
  FSM = (I,S,O,NO,NS)
- Realization as a binary system
  - coding of I, S, O.
- Input coding: $I \rightarrow \{0,1\}^n$
- Output coding: $O \rightarrow \{0,1\}^m$
- State coding: $S \rightarrow \{0,1\}^s$
- Binary combinational functions:
  NS: $S*I \rightarrow S$ or NS: $\{0,1\}^{s+n} \rightarrow \{0,1\}^s$
  NO: $S*I \rightarrow O$ or NO: $\{0,1\}^{s+n} \rightarrow \{0,1\}^m$
- Memory operation:
  using flip-flops to record
  - individual state code bits s
  - individual output bits m

-7.14-

7.2 Binary sequential systems

In the previous paragraph we got to know the flip-flop as a possibility for implementing the memory function of finite state machines. Now we can direct ourselves towards the realization of finite state machines. Again we want to realize our machine as a binary system. Here not only input and output values are presented by binary tuples, but also the machine states. Notice that this completely agrees with the use of the flip-flop as a binary memory element.

Thus we have (see figure 7.14) to deal with three value sets (I, S, and O) which must be coded. We consider the input coding, the output coding and the somewhat different state assignment. They result in two combinational binary functions, which we know how to realize. The memory function is realized by the individual bits of the s-tuple related to the state (which is stored in an equivalent number of flip-flops). The same is true for the output m-tuple. In a previous chapter we have seen that input and output coding have a large influence on the complexity of the realization of a binary function. In finite state machines the chosen state assignment has influence on the complexity of both the next state and the next output functions. It is extra important to choose a state assignment that produces a simple realization. Unfortunately there is no exact solution for this problem. In the 1990s, computer programs will be available which can make a relatively optimal state assignment. In this course we shall limit ourselves to getting acquainted with a number of frequently used state assignments, without bothering about optimization.



STATE CODE ASSIGNMENT (1)

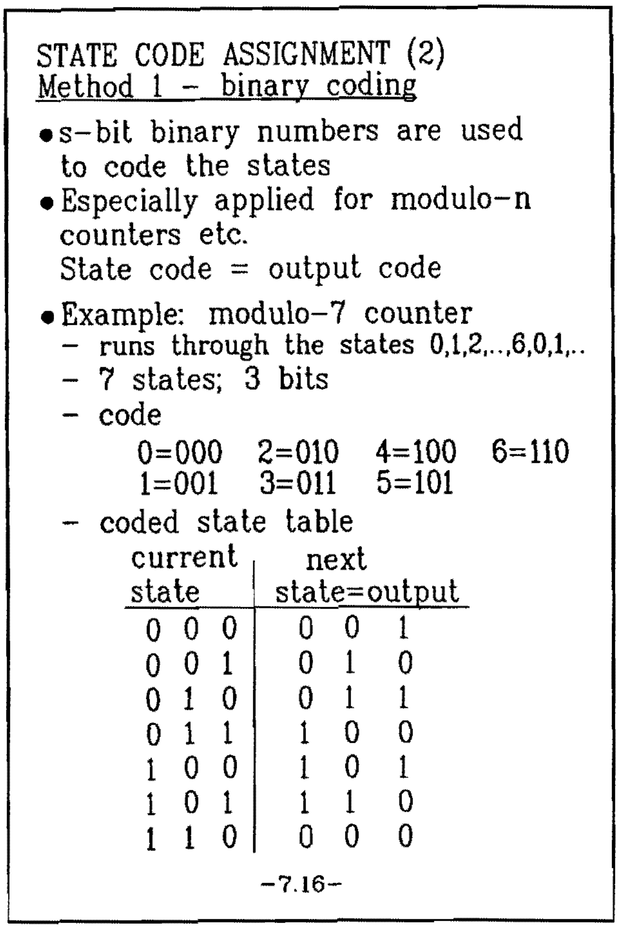
- State code
  The applied code has great influence on the NS and NO functions to be realized

- State minimization
  - Minimization of the number of states
    = number of elements in S
  - Minimization of number of flip-flops:

$$s = \lceil \log_2 \#s \rceil$$

17 states ⟶ 5 flipflops
16 states ⟶ 4 flipflops
9 states ⟶ 4 flipflops

-7.15-



STATE CODE ASSIGNMENT (2)
Method 1 - binary coding

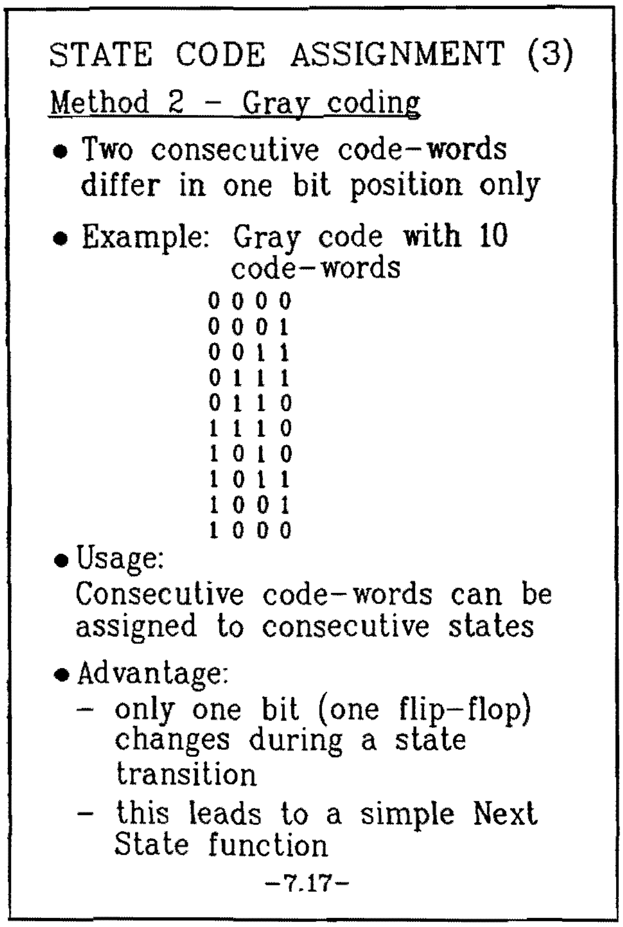
- s-bit binary numbers are used to code the states
- Especially applied for modulo-n counters etc.
  State code = output code
- Example: modulo-7 counter
  - runs through the states 0,1,2,..,6,0,1,..
  - 7 states; 3 bits
  - code
    0=000 2=010 4=100 6=110
    1=001 3=011 5=101
  - coded state table

current state			next state=output		
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0

-7.16-

Figure 7.15 also mentions state minimization as a method for reducing the number of states, i.e. the number of elements in S. Reducing the number of states can influence the complexity of the realization of the next state and next output functions. Also a smaller number of states can lead to a smaller number of flip-flops. However, as shown in figure 7.15, this reduction has a logarithmic nature; consequently possible simplifications are minimal. We shall not say anymore about state minimization in this course, except that in the 1990s programs will be available that can carry out this state minimization in combination with an optimal state assignment. Methods for state minimization will not be discussed here.

In state assignments we associate a binary tuple of a certain length with each element of the set of states S. We shall now discuss four methods for state assignment.
First in figure 7.16 binary coding is shown. Here we use s-bit binary numbers for coding the states. The number of bits s is equal to $\log_2(\#S)$, where #S denotes the number of elements in S. Such a code is frequently used in the modulo-n counter, where consecutive states are coded with consecutive binary numbers. The state code is generally equal to the output code, such that no separate next output function is necessary anymore. As an example we have coded in figure 7.16 the 7 states of a modulo 7 counter with three bits binary digits. We can then construct a coded state table being a truth table of our next state function.



STATE CODE ASSIGNMENT (3)

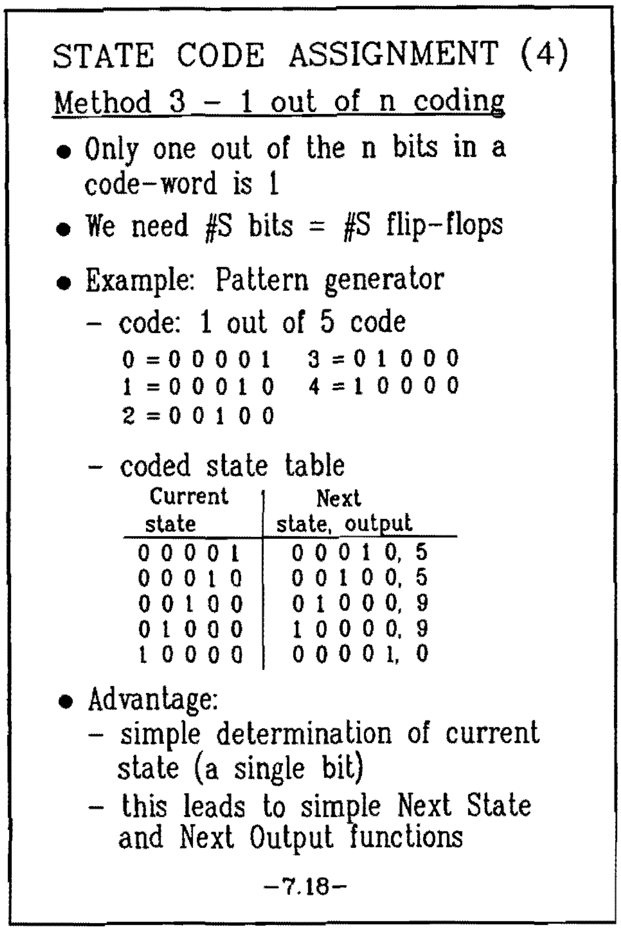
Method 2 - Gray coding

- Two consecutive code-words differ in one bit position only
- Example: Gray code with 10 code-words

```
0 0 0 0
0 0 0 1
0 0 1 1
0 1 1 1
0 1 1 0
1 1 1 0
1 0 1 0
1 0 1 1
1 0 0 1
1 0 0 0
```

- Usage:
  Consecutive code-words can be assigned to consecutive states
- Advantage:
  - only one bit (one flip-flop) changes during a state transition
  - this leads to a simple Next State function

-7.17-



STATE CODE ASSIGNMENT (4)

Method 3 - 1 out of n coding

- Only one out of the n bits in a code-word is 1
- We need #S bits = #S flip-flops
- Example: Pattern generator
  - code: 1 out of 5 code

```
0 = 0 0 0 0 1   3 = 0 1 0 0 0
1 = 0 0 0 1 0   4 = 1 0 0 0 0
2 = 0 0 1 0 0
```

  - coded state table

Current state	Next state, output
0 0 0 0 1	0 0 0 1 0, 5
0 0 0 1 0	0 0 1 0 0, 5
0 0 1 0 0	0 1 0 0 0, 9
0 1 0 0 0	1 0 0 0 0, 9
1 0 0 0 0	0 0 0 0 1, 0

- Advantage:
  - simple determination of current state (a single bit)
  - this leads to simple Next State and Next Output functions

-7.18-

As a second possible method for state assignment we introduce the Gray code in figure 7.17. In a Gray code, two consecutive code-words are constructed in such a way that they only differ from each other in one bit position. As we go from one code word to the following, only one bit changes. In figure 7.17 we show as an example a Gray code with 10 code-words. Check for yourself that this code indeed fulfils the above mentioned characteristics of the Gray code.

Problem (7.2)
Construct another Gray code with 10 code-words.

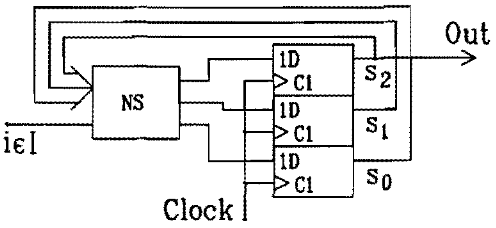
In state assignments we use the Gray code in the following manner: if we go from a certain state to another, we are considering *consecutive states*. We can now assign to these consecutive state consecutive code-words. The advantage is that in a transition form one state to another only one bit is the state code needs to be changed. Thus only one flip-flop needs to change value. Depending on the used flip-flop, this can lead to a simple next state function. We call the Gray code an *uni-variant* code, and we also speak about a uni-variant state assignment.

In figure 7.18 we introduce a third method of state assignment: the 1-out-of-n coding scheme. In such a code only one bit in a code-word is equal to 1. If our code-words have a length of n bits, we can in total make n code-words. To use this in our state assignments we need code-words with length of #S bits. This means that we also need #S flip-flops to store the state. The large number of required flip-flops is a disadvantage of the 1 out of n coding. As an example we have coded in figure 7.18 the states of our pattern generator with the 1 out of 5 code. The coded state table is also shown. Compare this table with the one given in figure 6.39. The 1 out of n code has the advantage that we can very simply determine what the current state is; we only need to observe which bit is 1. In combination with a programmable or-array, such as the one shown in figure 5.11, this leads to a simple realization of the next state and next output functions. The individual bits of the current state take the place of the minterms.

STATE CODE ASSIGNMENT (5)

Method. 4 - Coding "by inspection"

- Determine a code simplifying the realization of next state and next output functions
- Example-Electronic digit lock
  - Output coding:
    locked =0
    open =1 (only in state 4D)
  - S state bit(s) and output function should preferently be combined
  - Code assignment for $s_2 s_1 s_0$:
    0D=000 2D=011 4D=110
    1D=001 3D=010
    With the chosen coding the lock is only open if S_2 = 1



-7.19-

The fourth method of state assignment which is shown in figure 7.19 we have called "coding by inspection". This indicates that we shall not use a fixed code, but that we determine what an optimal code might be for each specific case. An optimal code must lead to a simple realization of the next state and next output functions. This is a *trial-and-error* "methodology".There are no fixed rules to give in order to find an optimal coding. Sometimes the used output coding can be indicative. We attempt to code the states in such a way that the next output function is simple. Frequently, we apply a variant of our canonical form shown in item 1 of figure 6.44, by trying to use bits of the state code for the output.

In other cases a simple realization of the next state function can be an objective. In figure 7.19 we have shown as an example of the first case, a state assignment of our electronic digit lock. Here we start from the given output coding: locked = 0 and open = 1. We know that the lock must only open if we go to state 4D, or the lock is only open if we are in state 4D. We can now arrange the state assignments in a way that we can use one bit of the binary state tuple for the output function. Then we do not need a separate next output function and we do not need separate flip-flops. With the code shown in figure 7.19 this is indeed possible. We see that now bit s_2 only becomes equal to 1 in state 4D. Thus this bit can also be used for the output function.

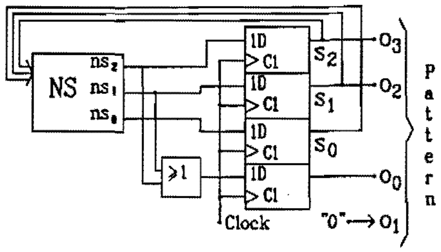
CODING BY INSPECTION - EXAMPLE

Pattern generator

- Binary output coding
  0=0000 5=0101 9=1001
- Choice of state coding
  0=000 2=011 4=100
  1=010 3=101
- Coded state table

State	Next state	Next Output	
$s_2 s_1 s_0$	$ns_2 ns_1 ns_0$,	$no_3 no_2 no_1 no_0$	
0 0 0	0 1 0,	0 1 0 1	5
0 1 0	0 1 1,	0 1 0 1	5
0 1 1	1 0 1,	1 0 0 1	9
1 0 1	1 0 0,	1 0 0 1	9
1 0 0	0 0 0,	0 0 0 0	0

- We observe that
  $no_3 = ns_2; no_2 = ns_1; no_1 = 0; no_0 = ns_2 + ns_1$
  Consequently there are no additional flip-flops for o_3, o_2 and o_1
- Realization



-7.20-

In figure 7.20 we have applied the coding "by inspection" to the state assignment of our pattern generator. Here the output values 0, 5 and 9 must be coded in binary. If we now chose the shown code for the 5 states of our pattern generator, then this leads to the coded state table shown in figure 7.20. From this table we see that the next output and next state functions are almost identical. We specially see that for the bits the following is valid: $no_3 = ns_2$, $no_2 = ns_1$, $no_1 = 0$, $no_0 = ns_2 + ns_1$. We see that this leads to a relatively simple realization.

Problem (7.3)

Give a realization of the next state function of this pattern generator using and-gates and or-gates.

We have now seen that there are different methods for state assignment. But they lead to one result: states coded as sets of binary tuples. Together with the input and output coding, this leads to the binary next state and next output functions. These functions produce as a result a binary s-tuple and a binary m-tuple, respectively. The individual elements, bits of these tuples, need to be clocked into their flip-flops. Actually, in general this cannot be done straight away. From the previous paragraph we know that each type of flip-flop has its own way of control. We thus have to deal with the input conditions of the used flip-flops. So a set-reset flip-flop will demand other requirements of the input values than a D flip-flop.

FLIP-FLOP TYPES (1)

- Next state function NS $\{0,1\}^{s+n} \rightarrow \{0,1\}^{s}$
- Next output function NO $\{0,1\}^{s+n} \rightarrow \{0,1\}^{m}$
- D flip-flop: $Q_{t+1}=D$
  - Input D = ns; upon active clock edge the new state is copied
  - NS and NO functions: no modification
- T flip-flop: $Q_{t+1}=T \oplus Q_t$
  - Input: T = s $\oplus$ns; upon active clock edge new state is determined
  - NS and NO functions: must be modified ("excitation functions")

Example: modulo-6 Gray code counter

s_2	s_1	s_0	ns_2	ns_1	ns_0	t_2	t_1	t_0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	1	0	1	0
0	1	1	0	1	0	0	0	1
0	1	0	1	1	0	1	0	0
1	1	0	1	0	0	0	1	0
1	0	0	0	0	0	1	0	0

$t_i = s_i \oplus ns_i$

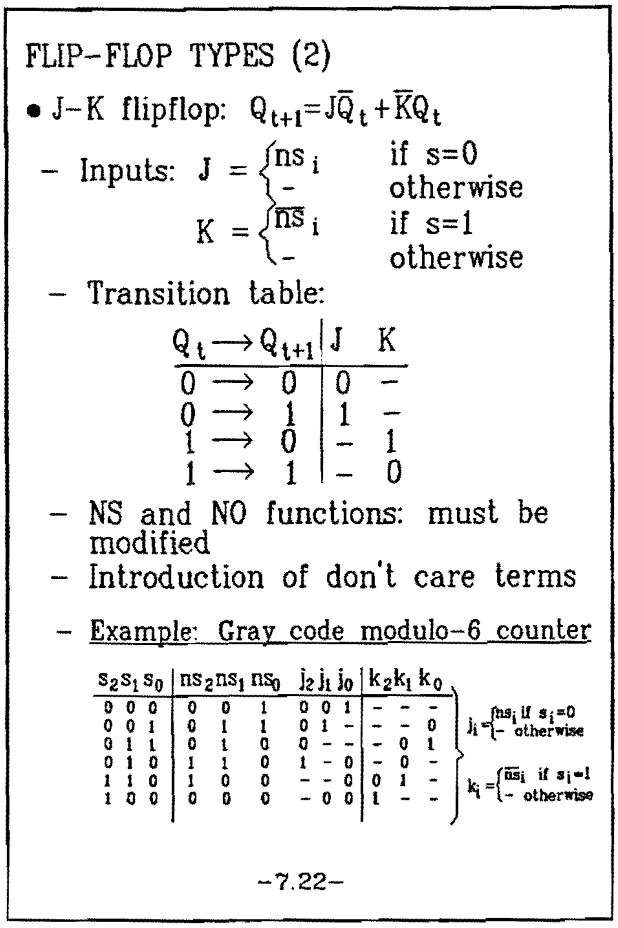
-7.21-

In figure 7.21 we first direct ourselves to the demands of a D flip-flop. We see from the characteristic equation that Q_{t1} = D; if we make the input D equal to a bit ns_i from the next state tuple, then the new state will be assumed on the active clock edge. In a D flip-flop we can thus always connect the binary coded next state. Modifications in the function ns or no are not necessary here. Because of these reasons we have only used D flip-flops in previous examples.

In figure 7.21 we also consider the demands of the T flip-flop. From the characteristic equation of this flip-flop we distinguish that the input condition for the trigger input T must be equal to $Q_t \oplus Q_{t+1}$. Actually, when used as memory for a bit of the state code, it is valid that Q_t is equal to the current state bit s and that Q_{t+1} represents the next state bit ns. Thus the condition for T must be s $\oplus$ ns. An adaption of the next state and next output function is necessary: we are considering the *excitation function*. Figure 7.21 finally shows a modulo-6 counter for a Gray code that contains the modification. We see first the coded state table with the current state bits s_2 and s_0 and the next state bits ns_2 to ns_0. The columns with a next state bit need now to be replaced by the shown columns for the trigger inputs of the three T flip-flops.

Problem (7.4)

Give a schematic realization of the Gray-code modulo-6 counter (in fig. 7.21). Realize the excitation function using a PAL.



FLIP-FLOP TYPES (2)

- J-K flipflop: $Q_{t+1}=J\bar{Q}_t+\bar{K}Q_t$
  - Inputs: $J = \begin{cases} ns_i & \text{if } s=0 \\ - & \text{otherwise} \end{cases}$

    $K = \begin{cases} \overline{ns}_i & \text{if } s=1 \\ - & \text{otherwise} \end{cases}$
  - Transition table:

$Q_t \longrightarrow Q_{t+1}$	J	K
$0 \longrightarrow 0$	0	-
$0 \longrightarrow 1$	1	-
$1 \longrightarrow 0$	-	1
$1 \longrightarrow 1$	-	0

  - NS and NO functions: must be modified
  - Introduction of don't care terms
  - <u>Example: Gray code modulo-6 counter</u>

s_2	s_1	s_0	ns_2	ns_1	ns_0	j_2	j_1	j_0	k_2	k_1	k_0
0	0	0	0	0	1	0	0	1	-	-	-
0	0	1	0	1	1	0	1	-	-	-	0
0	1	1	0	1	0	0	-	-	-	0	1
0	1	0	1	1	0	1	-	0	-	0	-
1	1	0	1	0	0	-	-	0	0	1	-
1	0	0	0	0	0	-	0	0	1	-	-

$j_i = \begin{cases} ns_i & \text{if } s_i=0 \\ - & \text{otherwise} \end{cases}$

$k_i = \begin{cases} \overline{ns}_i & \text{if } s_i=1 \\ - & \text{otherwise} \end{cases}$

-7.22-

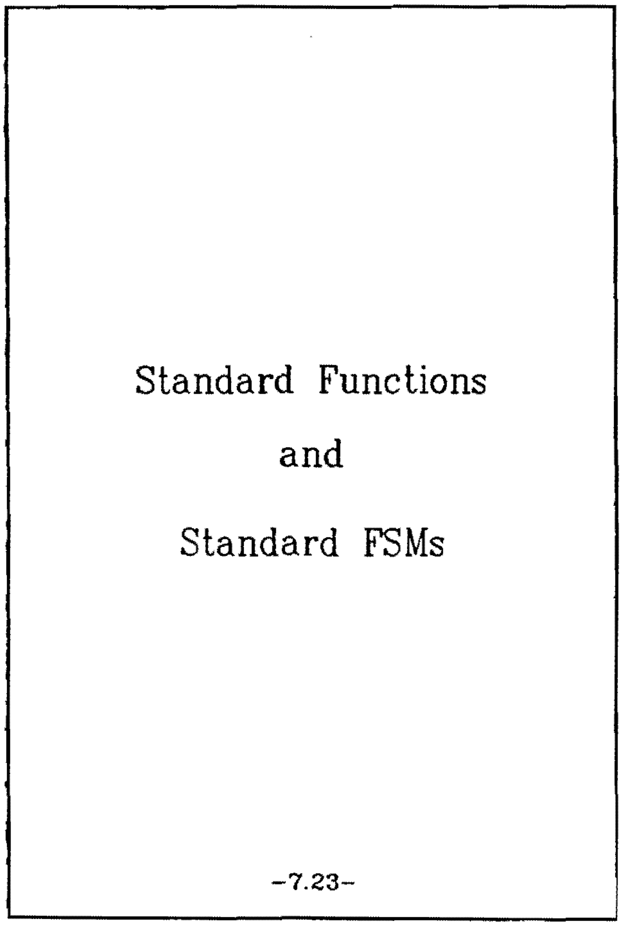
Finally we discuss in figure 7.22 the demands set by a JK flip-flop. Again our point is the characteristic equation. Here we consider again that Q_{t+1} represents the next state bit and Q_t represents the current state bit. The J input only has influence on the following state of the flip-flop if the current state is equal to 0. In the other cases the value of the J input is don't care. J must be equal to ns if s is 0. At the same time we conclude that the K input must be equal to ns' (not ns) if the current state s = 1. In the other cases the value of the K input is don't care. The necessary J and K inputs, related to the desired state transitions of the flip-flop are shown in tabular form. We see that the use of a JK flip-flop requires a basic adaption of the next state and next output functions. Also there are extra input don't care terms. In figure 7.22 we show what this means for the Gray code modulo 6 counter.

Now the next state columns must be replaced by the columns of the J and K inputs of our flip-flops. Instead of three binary next state functions we must now realize six binary excitation functions. The conclusion that it is therefore better not to use JK flip-flops in the realization of finite state machines is however not correct. The choice of the best flip-flop to use, strongly depends on the type of machine. JK flip-flops are well suited in counter-based finite state machines. In such cases, the use of the JK flip-flop will generally lead to simple excitation functions, simpler than ones of the other flip-flop types.

Problem (7.5)

The coded state table of our pattern generator is shown in figure 7.20. If we use JK flip-flops for the realization, how will the table of excitation functions look?

What do you notice about the binary output function no_0?

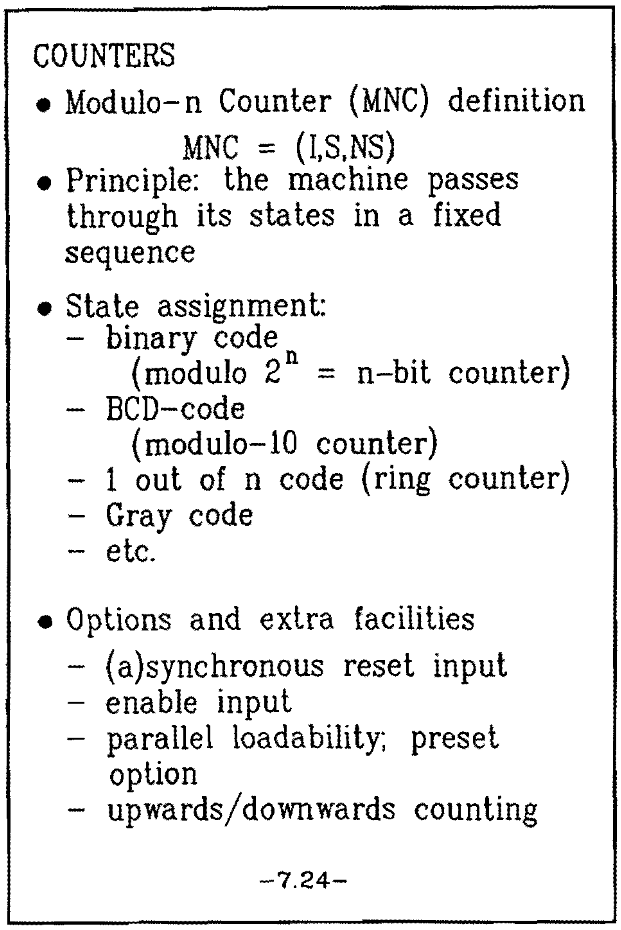


Standard Functions

and

Standard FSMs

-7.23-



COUNTERS

- Modulo-n Counter (MNC) definition

  MNC = (I,S,NS)
- Principle: the machine passes through its states in a fixed sequence
- State assignment:
  - binary code (modulo 2^n = n-bit counter)
  - BCD-code (modulo-10 counter)
  - 1 out of n code (ring counter)
  - Gray code
  - etc.
- Options and extra facilities
  - (a)synchronous reset input
  - enable input
  - parallel loadability; preset option
  - upwards/downwards counting

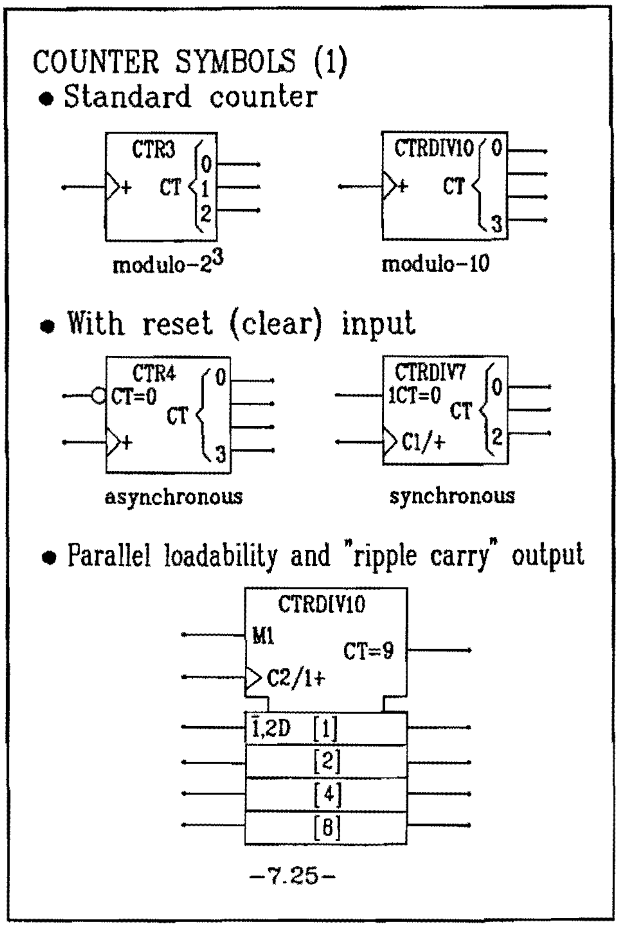
-7.24-

7.3 Standard functions; standard FSMs

In this paragraph we shall concentrate on the behaviour of two very frequently occurring types of finite state machine types: the counter and register functions. These functions are so frequently and generally used, that they are produced as standard building blocks. We shall explain the behaviour of some of these standard building blocks with the help of the common standard IEC symbols.

In figure 7.24 we first discuss the counter functions. We know that a counter is a special finite state machine with the characteristic that the states continually go through the same sequence, and that there is no separate next output function.

A counter with p states is called a modulo-p counter. Counters can be classified by the used state code. Generally, binary code is used for a modulo 2^n or an n-bit counter. If we have to deal with a modulo-10 counter, the use of BCD-code is more natural. If the 1 out of n code is used we are considering a ring or Johnson counter. Also other state codes such as the Gray code are possible. In addition, a counter can offer realizations with extra functionality. We have summarized the different possibilities in figure 7.24 under the heading "options". A counter can have a separate input to set it to 0. This is called a *reset* or *clear* input. A standard counter will increment the counter state by 1 for each clock pulse. Frequently this is not desired; we want to be able to stop and start this counting function with a separate input. Such an input is called *enable input.* Some counters offer the possibility to enter an initial state via separate inputs, i.e. to load it. Next there are counters that not only count in one direction, up, but can count in both directions, up as well as down.



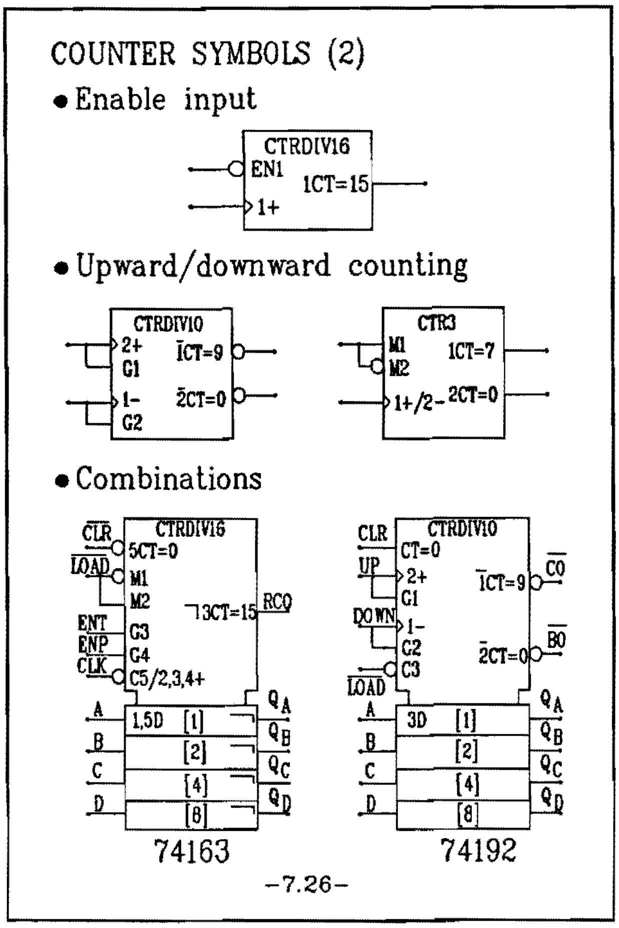
COUNTER SYMBOLS (1)

- Standard counter
- With reset (clear) input
- Parallel loadability and "ripple carry" output

-7.25-

In figure 7.25 we have shown the standard symbols for different realizations of counter functions. We begin with two standard counters. The left counter is an ordinary binary 3-bit counter, denoted by the name CTR3 (derived from the word "counter"). We see that a plus sign is placed at the edge-sensitive input. This "+" shows that the counter will make a positive counting step at each leading clock edge. The count value CT is available on the 3-bit output numbered from 0 to 2. At the right side we have shown a modulo-10 counter, denoted with the name CTRDIV10. The symbol is similar to the one of our binary counter. A reset or clear input is used to set the count value to 0. This can in principle be effectuated in two ways: asynchronously (independently of the clock), or synchronously (based on the clock command). Notice that in the synchronous implementation of CRTDIV7 the edge-sensitive input now has a double function. Firstly it serves as a command input for the clear function, expressed by the letters C1. Secondly this input has a counting function expressed by the pulse sign. Notice that the notation "1CT=0" at the clear input means that if this input is 1 at the following clock edge (*command dependency*) the counter CT is set to 0.

Finally we have in figure 7.25 drawn another counter where a counter value can be loaded via separate inputs. Because this function has influence on the behaviour of the individual flip-flops we must draw these flip-flops separately. This is shown in the lower half of the symbol; here the flip-flops are identified by the weight of the respective bits in the BCD-code. The upper half of the symbol forms the common control part. One input is used to choose one of the counter modes: "count" or "load". This is an input with the so-called *mode dependency*, expressed by M1. The edge-sensitive input has two functions. First there is a command dependency of C2, and next there is the counting function. With "1+" we express that the count function is only activated if the mode-input M1 = 1. Furthermore, we see in the first individual flip-flop (indicated with "$\bar{1}$,2D") that the flip-flop takes the value on the corresponding D input if input M1=0 and there is a positive clock edge. This counter has another output in the common control part. It is indicated with "CT=9" that this output is only 1 if the counter position = 9. Such an output is called a ripple-carry output and is meant to control the following counter. A following counter must than have the so-called enable input.



COUNTER SYMBOLS (2)

- Enable input
- Upward/downward counting
- Combinations

-7.26-

Such a counter is shown in figure 7.26. This counter has an input with an enable dependency, expressed with EN1. The used symbol by the edge-sensitive input shows that the counter functions if the internal value of this enable input is 1. Because of the inverter ball the external value on the input should be equal to 0. We see also that this counter has a ripple-carry output, which is equal to 1 if the counter position is equal to 15 and the counter is enabled. In this counter the individual bits are otherwise not available.

For counters which can count upwards as well as downwards, there are two possible implementations. The left counter in figure 7.26 has two edge-sensitive inputs. At the same time we indicate with the G dependency, that if this input is high the counting function of the other input is operational. At edges of the higher input the counter will count upwards, while at edges of the lower input the counter will count downwards. This corresponds to the plus and minus signs at the respective inputs. Notice also that there are now two ripple-carry outputs, one for each direction.

The right counter has only one edge-sensitive input, but also a separate mode-input. With "M1" and "M2" we express a single input, indicating whether mode M1 or mode M2 is active. From the symbol at the edge-sensitive input we read that mode M1 corresponds with upward counting and mode M2 with downward counting. This counter also has two ripple-carry outputs being activated depending on the chosen mode.

By combining a number of options we can compose complex counter functions. In the lower part of figure 7.26 the logical symbols for two practical counter circuits are shown. To the left we see the symbol of the 74163, a 4-bit binary counter with a clear function, a parallel-load option, 2 enable inputs and in addition to the 4-bit outputs, a separate ripple-carry output. We see from the symbol at the outputs that it makes use of master-slave flip-flops.

To the right we see the symbol of the 74192, a modulo-10 "up/down" counter. This counter can count upwards and downwards with the help of two separate edge-sensitive inputs. It also has a separate clear input and a possibility for the parallel loading of the counter. We also see two ripple-carry outputs.

MODULO-10 COUNTER (1)

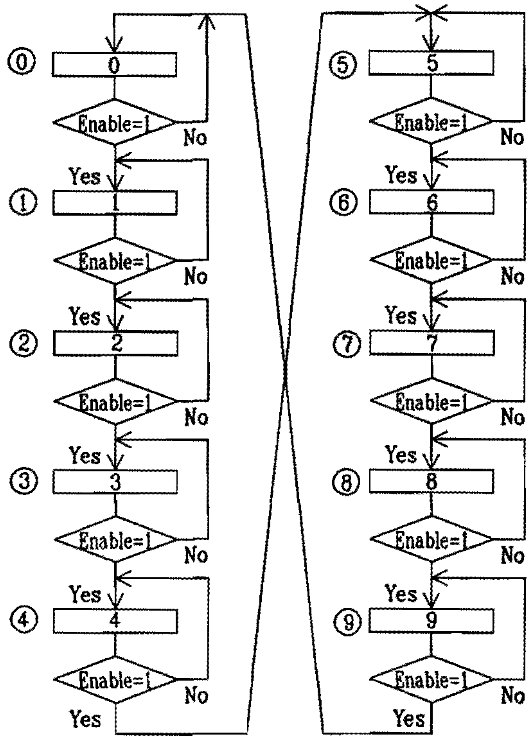
- Function:
  - modulo-10 counter
  - enable input
  - binary code
- Behavioural description

```
SYSTEM CTRDIV10;
TYPE RANGE = 0..9;
VAR  Clock : EVENT;
     Enable : BOOLEAN;
     Counter : RANGE;
BEGIN
  REPEAT
     WAIT_FOR_EVENT (clock)
     WAIT_FOR_INPUT (enable);
     IF (Enable)
        THEN Counter:=(Counter+1)MOD 10
  FOREVER
END.
```

-7.27-

MODULO-10 COUNTER (2)

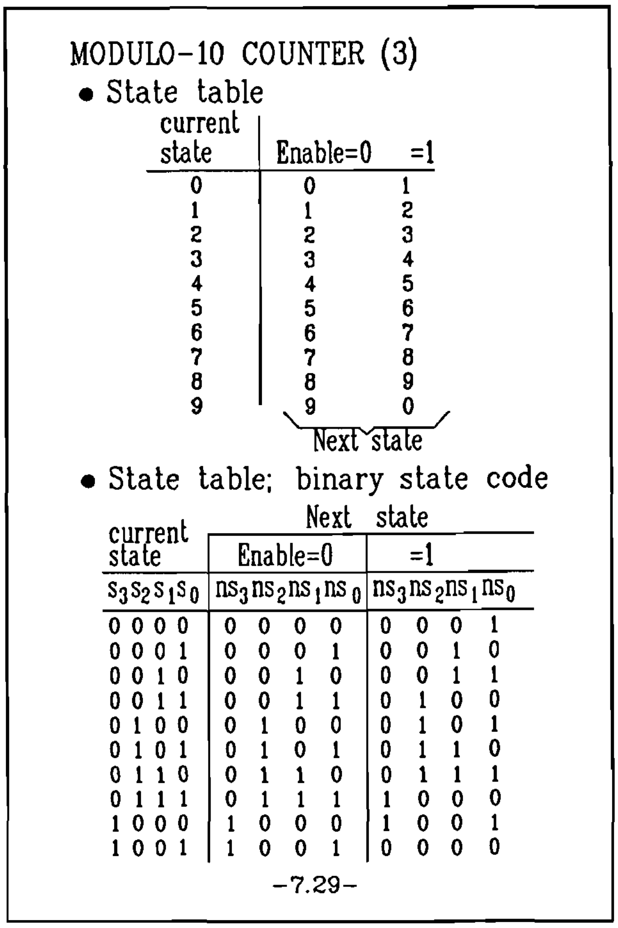
- Simplified ASM chart



-7.28-

Let us consider how to realize such a counter. In figure 7.27 we have formulated a design job. We want to make a simple modulo-10 counter with a separate enable input. The counter should use binary code. Our starting point is the behavioural description given in figure 7.27. Notice that the enable input appears in the IF statement within the REPEAT FOREVER loop. As long as "enable" is not true we do nothing; the counter position remains unchanged. When "enable" becomes true, the count will be incremented by 1 at each clock pulse.

From this behavioural description we derive the ASM-chart shown in figure 7.28.
Because we have to deal with a finite state machine where we can only have one output value per state, we can use the simplified ASM-chart. Here the next output values are written in the next states' boxes. Notice that the output values and states are equal to each other, as it should be in the case of a good counter. Notice also that we must test the value of the enable input in each state. Only if this value is equal to 1, we are allowed to proceed to the following state.



MODULO-10 COUNTER (3)

- State table

current state	Enable=0	=1
0	0	1
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7	7	8
8	8	9
9	9	0

Next state

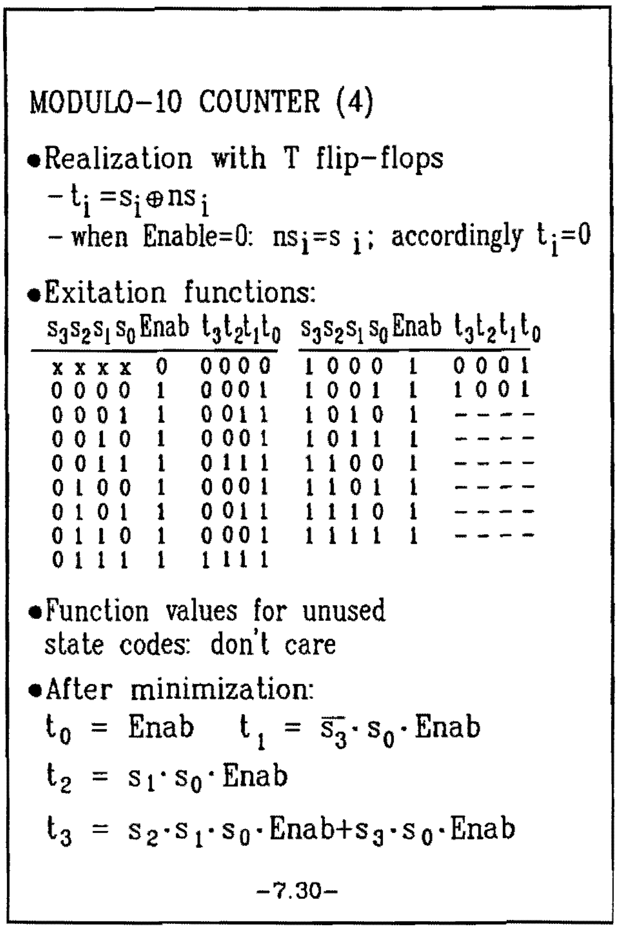
- State table; binary state code

current state	Next state	
	Enable=0	=1
$s_3s_2s_1s_0$	$ns_3ns_2ns_1ns_0$	$ns_3ns_2ns_1ns_0$
0 0 0 0	0 0 0 0	0 0 0 1
0 0 0 1	0 0 0 1	0 0 1 0
0 0 1 0	0 0 1 0	0 0 1 1
0 0 1 1	0 0 1 1	0 1 0 0
0 1 0 0	0 1 0 0	0 1 0 1
0 1 0 1	0 1 0 1	0 1 1 0
0 1 1 0	0 1 1 0	0 1 1 1
0 1 1 1	0 1 1 1	1 0 0 0
1 0 0 0	1 0 0 0	1 0 0 1
1 0 0 1	1 0 0 1	0 0 0 0

-7.29-

As a following step in our design process we must compose the state table. This is done in figure 7.29. We see that the 2 possibilities for the value of the enable input leads to two columns for the next state. Knowing that the next state is equal to the next output we have omitted the latter. The state table resulting from filling in the binary equivalent of the states is shown in figure 7.29. The current and next state are given by binary quadruples. This coded state table is in fact the truth table of our next state function.

Before we realize this function, we must determine which type of flip-flop to use. For the realization of a counter function, a T flip-flop or a JK flip-flop are often the most advisable types. Here we shall use the T flip-flop.



MODULO–10 COUNTER (4)

- Realization with T flip–flops
  - $t_i = s_i \oplus ns_i$
  - when Enable=0: $ns_i = s_i$; accordingly $t_i = 0$
- Exitation functions:

$s_3s_2s_1s_0$	Enab	$t_3t_2t_1t_0$	$s_3s_2s_1s_0$	Enab	$t_3t_2t_1t_0$
x x x x	0	0 0 0 0	1 0 0 0	1	0 0 0 1
0 0 0 0	1	0 0 0 1	1 0 0 1	1	1 0 0 1
0 0 0 1	1	0 0 1 1	1 0 1 0	1	- - - -
0 0 1 0	1	0 0 0 1	1 0 1 1	1	- - - -
0 0 1 1	1	0 1 1 1	1 1 0 0	1	- - - -
0 1 0 0	1	0 0 0 1	1 1 0 1	1	- - - -
0 1 0 1	1	0 0 1 1	1 1 1 0	1	- - - -
0 1 1 0	1	0 0 0 1	1 1 1 1	1	- - - -
0 1 1 1	1	1 1 1 1			

- Function values for unused state codes: don't care
- After minimization:

$t_0 = \text{Enab} \quad t_1 = \overline{s_3} \cdot s_0 \cdot \text{Enab}$

$t_2 = s_1 \cdot s_0 \cdot \text{Enab}$

$t_3 = s_2 \cdot s_1 \cdot s_0 \cdot \text{Enab} + s_3 \cdot s_0 \cdot \text{Enab}$

-7.30-

In figure 7.30 we show what this means. We know that the excitation function for the t input of each flip-flop is given by an exclusive-or function between the corresponding bits of the current state and the following state. By considering that the state does not change if the enable input has the value 0, and thus the following state is equal to the current state, we can write the first part of our excitation function in a brief but powerful manner. Indeed the t inputs must be equal to 0. In figure 7.30 we have translated the state table into a table of the excitation function. Notice the first rule of this table where the situation is sketched for "enable" = 0. The value of the state bits plays no role at this point. The t inputs of the flip-flops are 0. Furthermore, you can check for yourself that this table is derived from the state table of figure 7.29 by applying the previously discussed exclusive-or operation.

There is one more thing worth mentioning. When coding the states, codes occur that do not correspond to state symbols. Actually, these codes are legal and operational states for the binary implementation of finite state machines.

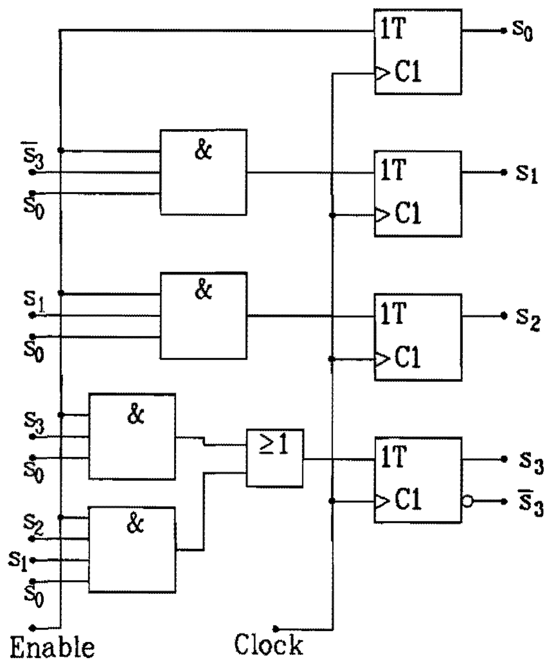
In principle, because the behaviour of the machine is not specified for these states, we can take the function value don't care. We assume that the machine only runs the specified states. Formally said this is correct reasoning. Actually in this world nothing is ideal, including the realization of a finite state machine. As a result of a disturbance or an error it can always happen that the finite state machine arrives in one of these undefined states. We always have to keep this possibility in mind. It would therefore be better to design the behaviour of the machine for the unspecified states, so that the machine can return to one of the specified states as quickly as possible.

We should at least convince ourselves that if the machine arrives at an unspecified state, it should proceed to one of the specified states. We shall not discuss this problem further.

We now have made a function table for our excitation function. This table can be minimized, which leads to the function (shown in figure 7.30) of the 4 inputs t_0 to t_3. Guided by these relations we can compose a schematic.

MODULO-10 COUNTER (5)

- Schematic



= next state determination of 74160

-7.31-

MODULO-8 COUNTER WITH CLEAR INPUT

- Behavioural description

```
SYSTEM CTR3_With_Clear;
TYPE RANGE = 0..7;
VAR Clock : EVENT;
    Clear : BOOLEAN;
    Counter : RANGE;
BEGIN
  REPEAT
    WAIT_FOR_EVENT(Clock);
    IF (Clear)
      THEN Counter := 0
      ELSE Counter :=(Counter+1)MOD 8
  FOREVER
END.
```

-7.32-

This is done in figure 7.31. In this case we have realized the excitation function with separate gates. Another possibility would of course be a realization with a PLA or PAL. This we shall leave to a number of exercises. In the schematic of figure 7.31 we notice that this coincides with the next state determination of the 74160, a synchronous 4-bit-counter.

Problem (7.6)

In figure 7.32 the behavioural description of a modulo-8 counter with a clear input is given. Give a realization of this counter. Make use of D flip-flops. Show that the use of T flip-flops or JK flip-flops is an advantage here.

Problem (7.7)
The behaviour of a modulo-8 up/down-counter can be described as follows:

```
TYPE        MODES : (Up, Down);
            COUNTSTATES : 0..7;
VAR         CountPulse : EVENT;
            Direction : MODES;
            CounterOutput : COUNTSTATES;
BEGIN
  REPEAT
    WAIT_FOR_EVENT(CountPulse);
    IF Direction = Up
      THEN CounterOutput := (CounterOutput + 1) mod 8
      ELSE CounterOutput := (CounterOutput - 1) mod 8
  FOREVER
END.
```

Questions:

a. *Describe the behaviour with an ASM-chart.*

b. *Make a state table.*

c. *Make a Grey code counter realization using T(oggle) flip-flops.*

REGISTERS (1)

- Function: memory
- Next State and New Output:

$$NS=NO : s_{t+1}=i_t$$

$$s_{t+1} \in S \ , \ i_t \in I$$

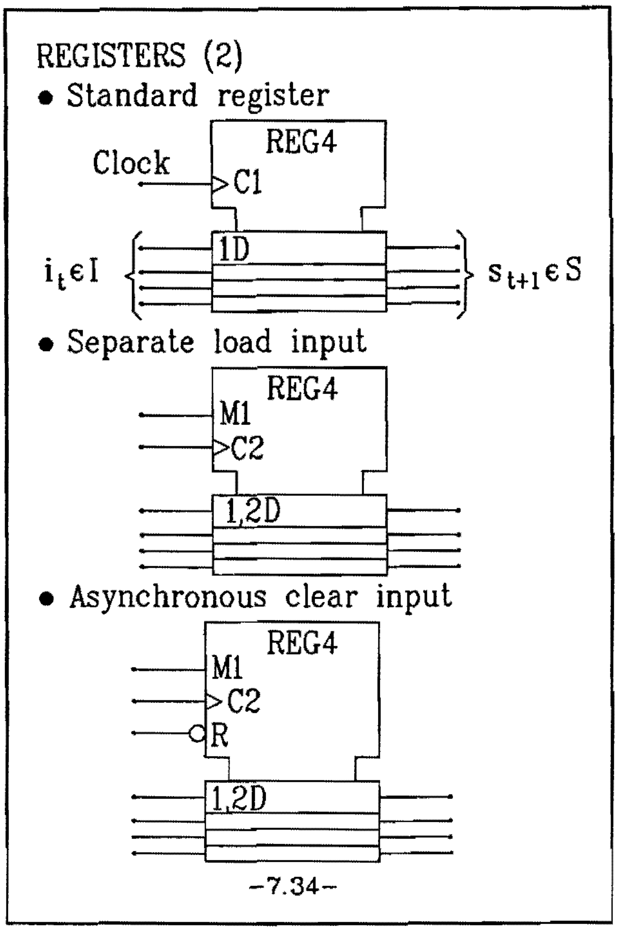
- Options and extra facilities
  - separate load input
  - (a)synchronous reset (clear) input
  - "3-state" outputs
  - shift register functions
    * shift left/right
    * serial input + serial output

-7.33-

As a second group of standard functions we shall discuss the register. In figure 7.33 is expressed that in this register function the emphasis lies on the memory function. We can save values in a register. Also in registers the next state function is in general equal to the next output. Formally we can describe the behaviour of a synchronous register with $s_{t+1} = i_t$, where t+1 stands for the time moment after the active clock edge and t for the one before. Upon each new clock edge a register will copy the value on its inputs.

Here we see that a register can be simply realized with the help of one or more flip-flops, depending on the number of bits we want to save. The combinational logic for the next state function is thus absent here. A register is formed by a set of flip-flops switching at the same clock edge.

Finally we can expand this standard register function with a number of extras. In figure 7.33 we have, among others, mentioned: a separate load input, a reset or clear input, "three (tri) state" outputs and shift-register functionality. The meanings of these terms will be further explained when discussing the corresponding building blocks.



REGISTERS (2)

- Standard register
- Separate load input
- Asynchronous clear input

-7.34-

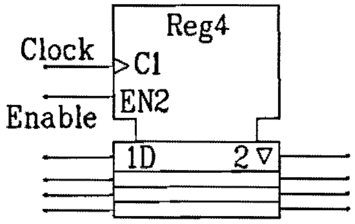
If figure 7.34 we have first drawn the symbol of a standard 4-bit register. Notice the name REG4 on top of the symbol. The symbol further contains a common control part, that is the highest part of the symbol. Here we find the edge-sensitive command input C1, where the clock is connected. Under this common part we find 4 rectangles as symbols for each of the individual flip-flops. Each flip-flop has a D input which is controlled by the common clock input. The operation is self-explanatory.

In many systems there is a central clock-generator that supplies all flip-flops with the clock signal. For a register that means that for each active clock edge new input data must be available. Of course this is not always desired. Frequently we want one register to keep the saved information for several clock pulses. This can be achieved with a register which has a separate load input.

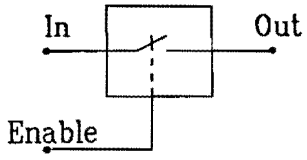
The symbol for such a register is shown in figure 7.34 (in the middle). Notice that a second input is added in the common control part, the mode-input M1. Furthermore, we see that the D input of the individual flip-flops are also controlled by this mode-input. These flip-flops will load new data when the mode-input M1 = 1, and the clock input has an active edge. If the mode-input M1 is low then the register will keep its old value; it will not change. In the bottom of figure 7.34 we have expanded this register with a reset (clear) input. Notice that the R input is independent of the clock input. Here we conclude that this clear input is obviously asynchronous. That is to say: when the external value of this input is equal to 0, all flip-flops will immediately assume the reset state, i.e. the 0 state without waiting for an active clock edge.

REGISTERS (3)

- 3-state outputs



- Enable = 0:
  - outputs are in their "3rd state" = high impedance
- Principle of 3-state output



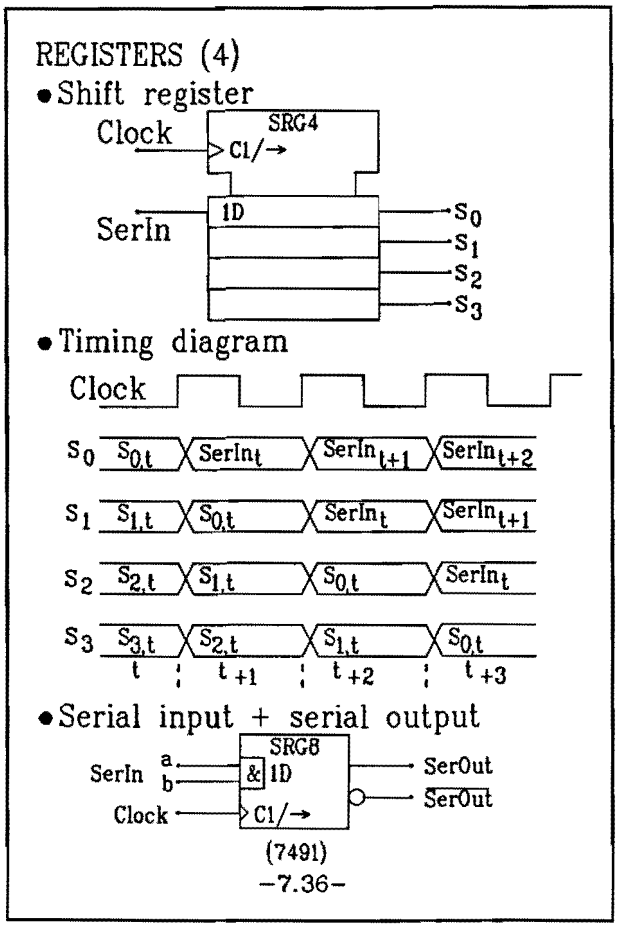
- Application: several devices with outputs connected together; only one of these may be enabled!

-7.35-

In figure 7.35 we have the symbol of a 4-bit register with the so-called "three (tri) state" outputs shown. A tri-state output is an output that in addition to the 0 state and the 1 state can be in a third state; this is where the name comes from. This third state is not a logic state, but is characterized by a very large output impedance. As shown in figure 7.35, for realization purposes, we shall consider it as a normal output which is connected in series with an electronically controlled switch. If the switch is closed the output is in the normal 0 or 1 state. When the switch is open the output is in the third state, the state of high impedance. In the symbol of the register the tri-state property is expressed by a triangle near the output of each individual flip-flop. The 2 near this rectangle shows that the tri-state of these outputs are controlled by another signal. In the common control block we see that this is the signal EN2, which we call the enable signal. This signal "enables" the output, i.e. lets the output work in its normal mode. When do we need such a tri-state output? In digital systems there are many situations where information of one of meany sources (e.g. registers) must be processed. If we now supply all these sources with tri-state outputs we can connect them all together. By enabling only the wanted source we avoid that several outputs will produce information at the same time and thus cause a short-circuit. The parallel connection of several sources and the selection of one of these sources is the purpose of the tri-state output. Also in computer systems three state outputs are frequently used at the interconnection of different subsystems.

REGISTERS (4)

- Shift register
- Timing diagram
- Serial input + serial output


Clock
SRG4
C1/→
SerIn
1D
s_0
s_1
s_2
s_3
Clock
s_0 $s_{0,t}$ SerIn$_t$ SerIn$_{t+1}$ SerIn$_{t+2}$
s_1 $s_{1,t}$ $s_{0,t}$ SerIn$_t$ SerIn$_{t+1}$
s_2 $s_{2,t}$ $s_{1,t}$ $s_{0,t}$ SerIn$_t$
s_3 $s_{3,t}$ $s_{2,t}$ $s_{1,t}$ $s_{0,t}$
t
t +1
t +2
t +3
SRG8
SerIn
a
b
&
1D
Clock
C1/→
SerOut
$\overline{\text{SerOut}}$
(7491)

-7.36-

In figure 7.36 we discuss the behaviour of a shift-register. On top we have drawn its standard symbol. Notice the letters SRG4 on top of the common control block. These letters are derived from the name Shift-ReGister. The shift-register is a register that can shift (move) its contents over one or more bit positions.

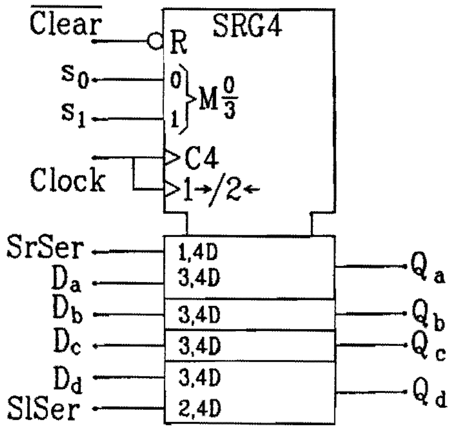
The behaviour is explained using the shown timing diagram. First we see that in time period t the values of the 4 outputs are shown. At the active clock edge the contents of the shift-register will be shifted one bit position. This means that the information of bit s_0 now arrives at the flip-flop of bit s_1. Similarly the s_1 bit shifts one position towards s_2 and at the same time the bit of s_2 shifts to s_3. The bit that was saved in bit s_3, is lost. The value of the serial input SerIn$_t$ is put in s_0. All of this is shown in time period t+1. On the following clock edge the whole process is repeated. Information from bits s_0, s_1, s_2 if shifted towards the bits s_1, s_2, s_3.

The new information of the serial input is put in s_0. This is shown in time period t+2. Now we return to our symbol. We see that the common control block has one input, the clock input, an edge-sensitive input with two functions.
First there is the combinational function C1 where the loading of the first flip-flop belonging to s_0 is controlled. Then we have the shift function which is expressed with the arrow. It is possible, at the same clock edge, to load new information in the first flip-flop s_0 and to shift the old information over 1 bit position.

We can imagine that there are many variants of the shift-register. We shall not discuss these in this course. We want to give some attention to a shift-register type, where not all output bits are available. The symbol of such a shift-register is shown in figure 7.36 (bottom). Notice that here we are discussing an 8-bit shift-register. The shift-register has an edge-sensitive clock input, 2 serial inputs combined by an "and" function, a serial output SerOut and its complement. The serial output corresponds to the output of the highest bit position. In our previous example that was s_3. We notice that the given device symbol is a 7491.

REGISTERS (5)

- 4-bit universal register with bidirectional shift option



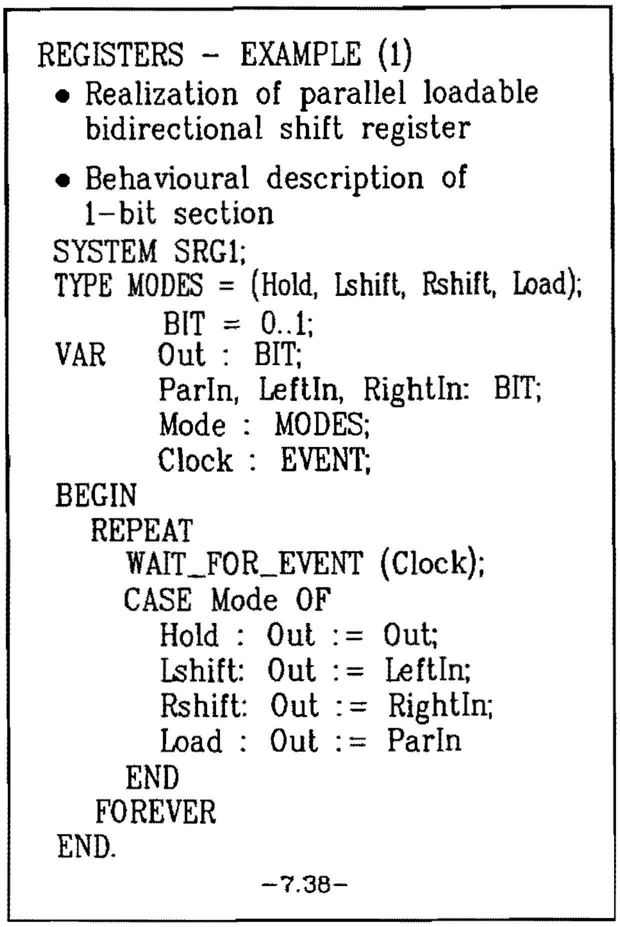
(=74194)

Mode: 0 = Hold,
1 = Shift Right
2 = Shift Left
3 = Parallel Load

-7.37-

Combining a number of items we arrive in figure 7.37 at the symbol of a 4-bit universal register, where we can shift the data in two directions (bidirectional). The upper part of the symbol represents the common control part. There we have the clear input that asynchronously puts the flip-flops in their 0 state. Then we have the two mode-inputs, expressed with s_0 and s_1. With M0/3 we show that using these two inputs, 4 different modes can be selected. These modes are: hold, shift right, shift left and parallel load. As a fourth input we have finally the clock input, an edge-sensitive input with several functions. There is first the command dependency C4.

Next we see that in mode 1 and 2 a shift operation is performed in one of both directions. Traditionally these directions are called "right" and "left". The lower part of the symbol shows the 4 flip-flops. Notice that the D inputs of these flip-flops only function in mode 3 and on an active clock edge. The higher and lower flip-flops both have an extra input, "shift right serial" SrSer for the highest flip-flop which is active in mode 1, and the input "shift left serial" S1Ser for the lower flip-flop which is active in mode 2. Note that the symbol shown in figure 7.37 is the 74194 building block.



REGISTERS - EXAMPLE (1)

- Realization of parallel loadable bidirectional shift register
- Behavioural description of 1-bit section

```
SYSTEM SRG1;
TYPE MODES = (Hold, Lshift, Rshift, Load);
       BIT = 0..1;
VAR    Out : BIT;
       ParIn, LeftIn, RightIn: BIT;
       Mode : MODES;
       Clock : EVENT;
BEGIN
  REPEAT
    WAIT_FOR_EVENT (Clock);
    CASE Mode OF
      Hold : Out := Out;
      Lshift: Out := LeftIn;
      Rshift: Out := RightIn;
      Load : Out := ParIn
    END
  FOREVER
END.
```

-7.38-

We now want to briefly address the question of how such a shift-register can be realized. In figure 7.38 an example has been worked out. The example shows the realization of a parallel loadable bidirectional shift-register. Notice that we have a small problem with shift-registers. A shift action cannot be easily described on a high symbolic level. Indeed shifting has effect on bits: on binary coded signals in binary systems. These we can not easily translate to operations on value sets or on symbol sets. The fact that we cannot give a higher symbolic description of the bottom binary building block layer is not a shortcoming of the general theory, but is to be expected.

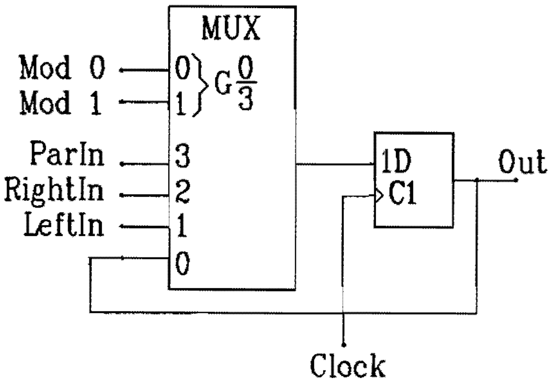
We can give a good behavioural description of a bit section of a shift-register, however. This is done in figure 7.38. Notice that we have first defined the first 4 modes in one type declaration. We have also given the value set of inputs and outputs in the BIT type. Subsequently we have declared the necessary variables for the inputs and outputs. Notice that this is a synchronous system: we have a clock input of type EVENT. The behaviour of one bit section can now be described as follows: we wait for an active clock edge. What has to be done afterwards is depending on the mode. In the Hold mode the value of the output Out must remain unchanged. In the Lshift mode the output is equal to the value on the serial input LeftIn. Similarly in the Rshift mode the output is made equal to the value on the RightIn input. Finally, if we are in the Load mode the output value is obtained by loading the value on the parallel input ParIn. This is the behaviour of a 1-bit shift-register. We could now sketch an ASM-chart based on this behavioural description and make a state table. In this case this is completely unnecessary.

REGISTERS - EXAMPLE (2)

- CASE statement = multiplexer behaviour
- Coding of input

Hold = 00   Rshift = 10
Lshift= 01   Load = 11

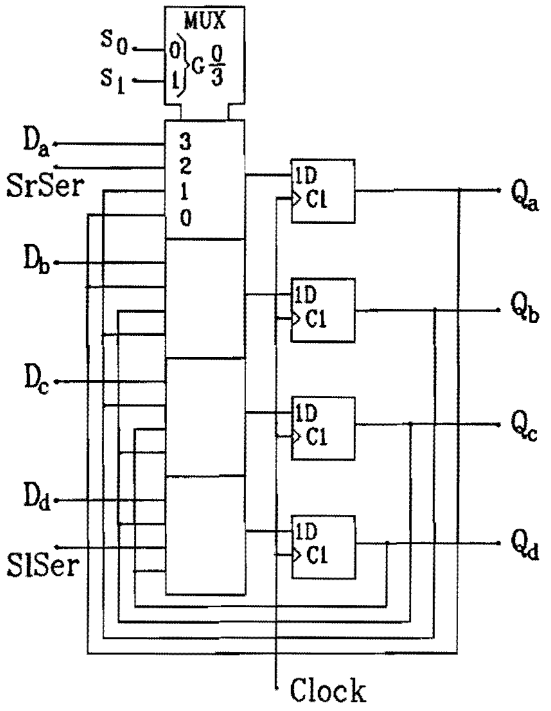
- Schematic



-7.39-

REGISTERS - EXAMPLE (3)

- Extension to 4 bits



-7.40-

If we look at the behavioural description again we recognize in the CASE statement the description of a multiplexer (figure 7.39). We could include this behavioural description in a separate procedure. What remains is the behaviour of a simple register, such as we have seen in the beginning of the previous chapter. The system is then composed of a multiplexer followed by a 1-bit register. In figure 7.39 we see that we still have to consider the coding of the modes. With the coding shown in figure 7.39 we find the illustrated realization. Notice the correspondence of the output Out to input 0 of the multiplexer. Here the Hold mode is realized. Furthermore, we still have three other data inputs for the parallel load mode, the right shift mode and the left shift mode. The use of these inputs can only be demonstrated if we expand our shift-register to several bits.

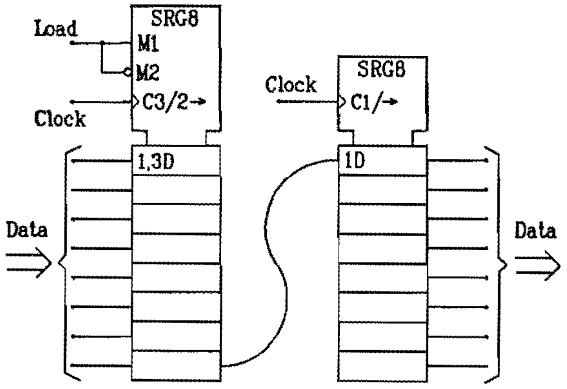
In figure 7.40 we have done this for an expansion to 4 bits. Notice that we have now 4 times a 4-to-1 multiplexer. Furthermore, we have brought the notation of the inputs in correspondence with the symbol shown in figure 7.37.

On top we first see the 2 inputs s_0 and s_1 for the selection of the modes. Further we see that inputs 0 of the 4 multiplexers are connected with the outputs of the corresponding bits of the flip-flops. The inputs 1 are the inputs for the left shift mode, and these are related to the flip-flops that have a lower position. Notice that for the last multiplexer input there is no lower flip-flop anymore. This is the shift left serial input. The multiplexer inputs 2 form the inputs of the right shift mode. These outputs are connected with the next higher positioned flip-flops. The input of the highest multiplexer forms the shift right serial input. The inputs 3 of the multiplexers form finally the 4 parallel load inputs. Notice that only with an addition of an asynchronous clear input this could be an implementation for the 74194.

REGISTERS - EXAMPLE (4)

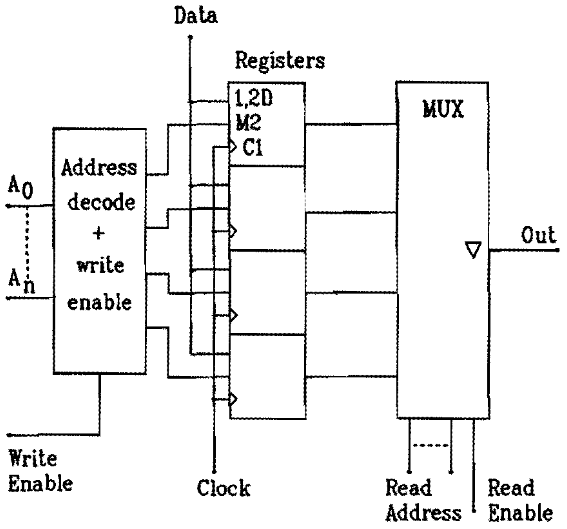
- Application:

  Serial communication



-7.41-

REGISTER FILE



-7.42-

Using figure 7.41 we want to discuss an important application of the shift-register: serial bit-communication between two systems, for example a computer and a terminal. Between both of these systems information must be exchanged. This information generally comprises a series of symbols coded as binary tuples. Such a symbol can for example consist of 8 bits. These symbols can now be exchanged using eight *parallel* wires, 1 bit per wire. This is a good possibility if the speed of information exchange has to be high. The disadvantage is a result of speed differences in the different wires: the distance to be crossed can only be relatively short. In *serial* communication the information is exchanged along a single wire. Here we load the symbol at the sender side (in parallel) into a shift-register and transmit it in serial, bit by bit, to the other side. At the receiver side these bits are put behind each other in a shift-register, after which the 8 bits of the symbol are available.

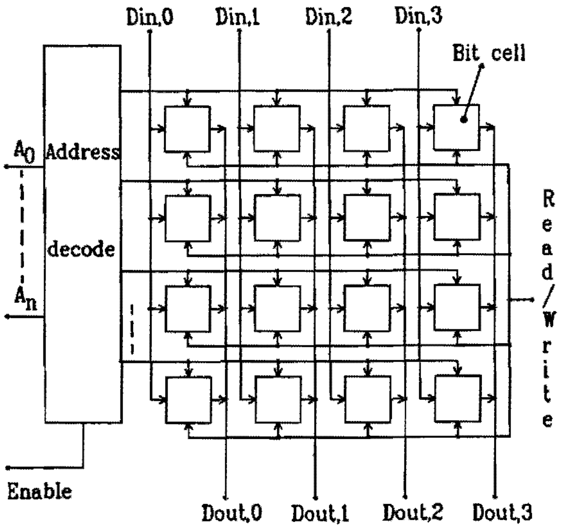
We have obtained a communication method via one wire. For this purpose we can use the world-wide telephone network or the future ISDN-network.

Finally we should have made some regulations to be sure that the sender and the receiver run exactly at the same speed; this is beyond the scope of this course, however.

In figure 7.42 we have expressed that we also can combine several registers (each of them may be several bits wide) in a register file. Several implementations of such register files are possible. In figure 7.42 we have drawn an implementation that allows us to write new data into a register and read another register at the same time. To do that it is necessary to provide 2 separate addresses. A writing address at the left side, indicating the register where we want to write data, and a read address at the right side to select the register that we want to read. We have used a multiplexer for the selection of reading the register. Notice that we have given this multiplexer a three-state output, so that we can connect several registers in parallel.

With the read enable input we can select the wanted register to be read out.

RANDOM ACCESS MEMORY (RAM)



-7.43-

A register file generally contains hundreds of registers. If we need to save more information we should consider memories. In figure 7.43 the principle scheme of a read/write memory, a random access memory (RAM), is shown. Such a circuit comprises a large number of identical flip-flops called *bit cells*, grouped in rows. By offering an address to the address decoder we can select such a row.

Each address selects a *word* in the memory. The bits in the selected word can be read or written. Compare this with the ROM discussed in the previous chapter. We notice that at this moment RAM ICs with 16 million bits are obtainable. This number is doubled each year.

7.4 Summary

In this chapter we considered the realization of finite state machines. For this realization we first need a memory function. We have seen that we can use flip-flops for this. The flip-flop is a bistable circuit, i.e. a circuit with two stable states. As an example we have seen the set-reset latch. For our purpose we need a flip-flop with a clock input. To this end we considered the edge-sensitive flip-flop or the master-slave flip-flop. Next we have distinguished different types of flip-flops: the D flip-flop, the T flip-flop and the JK flip-flop.

With the help of these types of flip-flops we can realize finite state machines. We have seen that next to the memory function we still must realize two combinational binary functions. An important new aspect here is the state assignment or the code that we use for the states of a machine. We have discussed four methods of doing that. We have noticed that none of the four methods can produce an optimal result in all cases. There is no known general methodology for doing this. The replacement of the next state and next output functions by the excitation function of the flip-flop is important for the realization.

In the second part of this chapter we have directed ourselves to the implementation of two standard functions, the counter function and the register function. We have discussed various options and given examples of the used symbols. Also we have given realizations of both types of functions. In the case of the register functions we introduced the grouping of registers in a register file, and the following step, the use of a random access memory.

Appendix A

Examinations

TECHNISCHE UNIVERSITEIT EINDHOVEN

TENTAMEN DIGITALE SYSTEMEN II
5A010

TOTAL: 8 sheets
of which 4 answersheets

Saturday 10 november 1990
time: 09.00 - 12.00 hours

SHEET 1

Questions should be answered on special added answersheets. You only have to hand in these answersheets.

1. Convert the next expressions to the form given, where for every x an inverted or normal variable should be substituded. Fill in your answers on the place provided in the answersheets.

   a) a + bcd = (x+x)(x+x)(x+x)

   b) a(b+c') + (b+c)' = xx + xx

   c) (ab + a'c + b'c)' = (x+x)x

   d) ab ⊕ ac = xxx + xxx

   e) ((a'+b)' + (a+b')')' = xx + xx

2. The behaviour of a combinational function can be described in the next way:

```
TYPE FUNC = (PLUS1, PLUS2);
VAR  In, Out : 0..7;
     Command : FUNC;
BEGIN
  IF Command = Plus1
    THEN Out := (In + 1) MOD 8
    ELSE Out := (In + 2) MOD 8;
END.
```

   It is decided to realise this function as a 3-iterative circuit. Answer the next questions on the places provided in the answersheets.

   a) Draw on the place provided in the answersheets a schematic diagram of the architecture of this 3-iterative circuit. Mention explicitly what signals with what set of values are on the different inputs and outputs.

   b) If for the values of the variable *Command* the next coding will be used:
   PLUS1 = 0, PLUS2 = 1
   then give in a truth table on the place provided the function of each of the blocks in your diagram.

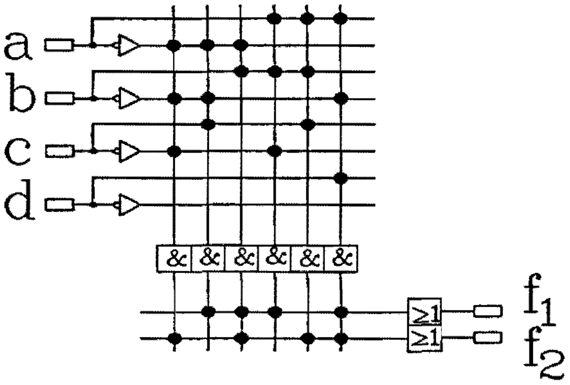
TECHNISCHE UNIVERSITEIT EINDHOVEN

TENTAMEN DIGITALE SYSTEMEN II
5A010

TOTAL: 8 sheets
of which 4 answersheets

Saturday 10 november 1990
time: 09.00 - 12.00 hours

3. In the next figure 2 functions f_1 and f_2 are realised with a PLA with 4 inputs and 2 outputs.



Answer the next questions in the place provided on the answerforms.
For the numeric values of the minterms it is assumed that a has the highest weight and d the lowest.

a) Calculate for both functions f_1 and f_2 the sum of minterms form, and put these in numeric format on the place provided in the answersheets.

b) The same question for the product of maxterms form of both functions.

c) Calculate for both dual functions f_{1d} and f_{2d} the sum of minterms form, and put these in numeric format on the place provided in the answersheets.

d) In the answersheets you will find the same PLA, but now unprogrammed. Draw the programming information for the PLA to realise both functions f_{1d} en f_{2d} in the same time.

4. A counter for which the state table is given below, can function as a modulo-5 or modulo-6 counter under control of an input m.

	m	
state	0	1
0	1	1
1	2	2
2	3	3
3	4	4
4	0	5
5	1	0

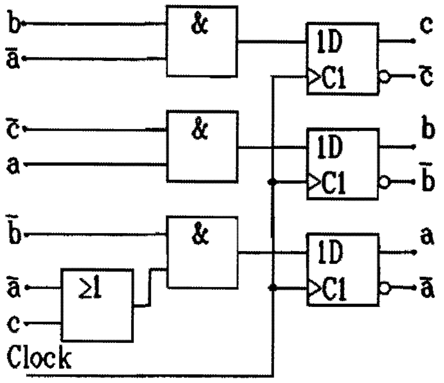
(continued on next page)

4. continued

In realising the counter, use is made of the binary code, while the flipflops are of the J-K type. In determing the numeric values of the minterms it is assumed that m has the highest weight, followed by the most significant bit of the state code, etc.

Write down in the place provided on the answersheets as a sum of minterms the functions needed on the J and K inputs of the flipflops. Use underlining to indicate the minterms that are don't care.

5. In the next figure a schematic of the realisation of a finite state machine is drawn.



Answer in the places provided on the answersheets the next questions:

a) Determine for this machine the consecutive states and write these down in the state table given.

b) After some clockpulses the circuit will get into a cycle. How many states contains this cycle?

6. On the next page the behaviour description of a finite state machine with 4 states, 1 input and 1 output is given. On one of the answersheets you will find a skeleton ASM-chart for this machine. Questions:

a) Complete the ASM-chart for this machine.

b) Write down in the place provided what values the output *Out* can have in the different states.

c) Fill in the state table of this machine that is given in one of the answersheets.

6 continued.

```
TYPE MODI  = (M1, M2);
    STATES = (S0, S1, S2, S3);
    OUTVAL = 0..7;
VAR  Clock : EVENT;
    State : STATES;
  Out   : OUTVAL;
    Choice: MODI;
BEGIN
  State := S0;
  Out := 0;
  REPEAT
    WAIT_FOR_EVENT( Clock);
    CASE State OF
      S0:  BEGIN
             IF Choice = M1
               THEN State := S1;
             Out := 1;
           END;
      S1:  BEGIN
             State := S2;
             IF Choice = M2
               THEN Out := 2
               ELSE Out := 7;
           END;
      S2:  BEGIN
             IF Choice = M1
               THEN State := S3
               ELSE State := S0;
             Out := 4;
           END;
      S3:  BEGIN
             State := S1;
             Out := 1;
           END
    END (* CASE *)
  UNTIL FOREVER
END.
```

TECHNISCHE UNIVERSITEIT EINDHOVEN

TENTAMEN DIGITALE SYSTEMEN II
5A010

TOTAL: 8 sheets
of which 4 answersheets

Saturday 10 november 1990
time: 09.00 - 12.00 hours

ANSWERINGSHEET 1

ANSWERSHEETS
NAME:______________________________

I.D. NUMBER:________________________

1. a)

   b)

   c)

   d)

   e)

2. a) space for drawing the schematic diagram

TECHNISCHE UNIVERSITEIT EINDHOVEN

TENTAMEN DIGITALE SYSTEMEN II
5A010

TOTAL: 8 sheets
of which 4 answersheets

Saturday 10 november 1990
time: 09.00 - 12.00 hours

ANSWERINGSHEET 2

ANSWERSHEETS
NAME:____

I.D. NUMBER:________________________________

Answer 2 continued

b)

Command In __	*Out* __

3. a) $f_1 = \Sigma$____

$f_2 = \Sigma$____

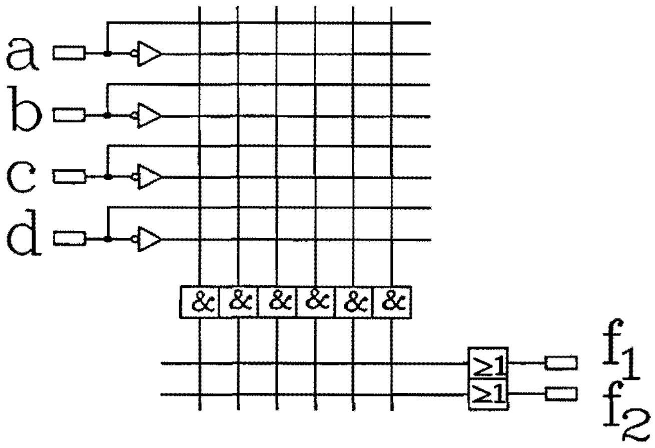
b) $f_1 = \Pi$____

$f_2 = \Pi$____

c) $f_{1d} = \Sigma$____

$f_{2d} = \Sigma$____

d)



TECHNISCHE UNIVERSITEIT EINDHOVEN

TENTAMEN DIGITALE SYSTEMEN II
5A010

TOTAL: 8 sheets
of which 4 answersheets

Saturday 10 november 1990
time: 09.00 - 12.00 hours

ANSWERINGSHEET 3

ANSWERSHEETS
NAME: ______________________________

I.D. NUMBER: ______________________

4. For the most significant state bit must hold:

J = Σ ______________________________

K = Σ ______________________________

For the next state bit must hold:

J = Σ ______________________________

K = Σ ______________________________

And for the least significant state bit must hold:

J = Σ ______________________________

K = Σ ______________________________

5. a)

current state c b a	next state c b a
0 0 0	
0 0 1	
0 1 0	
0 1 1	
1 0 0	
1 0 1	
1 1 0	
1 1 1	

b) The cycle contains ________ states.

EXAMINATION "DESIGN OF DIGITAL SYSTEMS"

Wednesday, 1991-03-13; 8.45 - 11.45 hours

Only the survey "Design of Digital Systems" (volume 1, 2 and 3) is allowed during examination.

1 Switching functions

Show by means of switching algebra and postulates the validity of the following. All subproblems have 3 variables a,b and c; concerning numerical representation of minterms and maxterms, a has the highest weight and c has the lowest one.

a. $ab + a'c + bc = ab + a'c$

b. $(a \oplus b)' = a \oplus b'$

c. $(a'(b + c'))' \cdot (a + b' + c) \cdot (a'b'c')' = a + b'c$

d. $a + b'c = \Pi(0,2,3)$

e. $\Sigma\,(0,1,2,4,5,6,7) = a + b' + c'$

2 Iterative combinational networks

Consider a system MulDiv2 with:

inputs	Din	∈ {0..7}	
	Mode	∈ {Mul1,Mul2,Div2}	
output	Dout	∈ {0..7}	
function	F:	Dout = Din	if SEL = Mul1
		Dout = (Din * 2) MOD 8	if SEL = Mul2
		Dout = (Din DIV 2)	if SEL = Div2

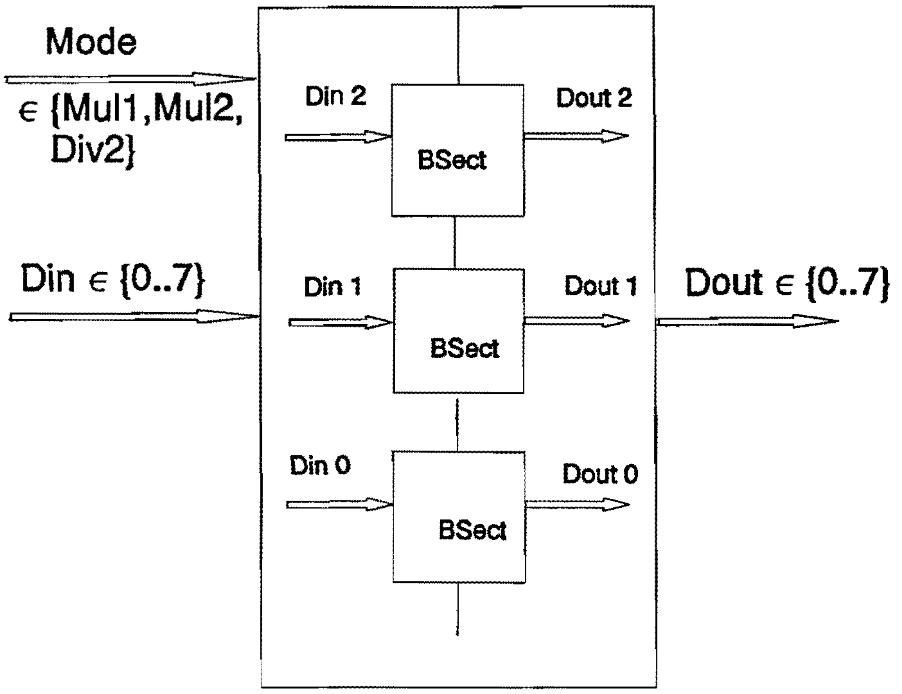
a. Make a behavioural description in Pascal.

MulDiv2 should be realized as a binary system, with Din and Dout coded as binary numbers.

b. Show with Din = 1, 2, 3, 5 that:

- Mode = Mul2 corresponds with a shift left operation
- Mode = Div2 corresponds with a shift right operation

MulDiv2 should be realized as a 3-iterative network with the following structure:



The dotted lines show possible interconnections between the 3 bit-sections BSect.

c. Make a more detailed drawing of the system, showing all interconnections between the bit sections. Show also how the Mode input is connected within the system.

   The internal structure of the BSect sections should *not* be included.

d. Make a behavioural description of BSect as a Pascal function "BSect":

```
    FUNCTION BSect (Din: ...., ...., Mode: ...): ...;
    ..................
```

   Make also a drawing of BSect (*not* the internal structure) with named input and output lines.

e. Make a behavioural description in Pascal of the whole system, making use of the Pascal function declared in (d).

3 Counter realization

One of the EII students has a girl friend. He phones her quite frequently, but forgets her telephone number (which is 462310 by the way) all the time. In order to save the work of searching for the number and dial it each time, he decides to construct an auto-dial circuit. As a part of this circuit he needs a special-purpose counter. It should have binary coded outputs; the counting sequence should correspond to the telephone number, and when finished, the counter should again return to the first digit in the number, being ready to start all over again for a next time. If the counter - due to malfunctioning - enters an unused state, this is viewed as an error situation, and the counter should be frozen.

Your colleague asks you to help him with the design of this counter. Of course you agree; you like that kind of work and it also gives you some training for the examination of "Design of Digital Systems"!

a. Construct a coded state table.

b. The counter should be realized using T(oggle) flip-flops. Add to the state table in (a) exitation vectors $t_2t_1t_0$ for the T flip-flops.

c. Write the Boolean functions for t_2, t_1 and t_0.

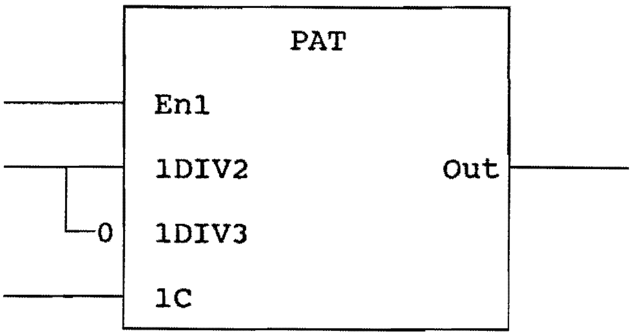
d. Realize the exitation functions t_2, t_1 and t_0 using the PLA on answer sheet 3d. Indicate signal names on all used inputs and outputs.

e. Discuss a possible use of Multiple Output Mimimization (MOM) to t_2, t_1 and t_0.

f. Would you recommend the use of a PAL (as opposed to a PLA) for this problem? Explain.

4 Finite State Machine

A one-bit pattern generator PAT has the following IEC-like symbol:



NB! The digits in the names DIV2/DIV3 are parts of their names and do not indicate any dependencies.

PAT is defined by the following behavioural description:

```
SYSTEM PAT;
TYPE States = (Low,High1,High2);
VAR           En,DIV2,DIV3:  BOOLEAN; { Inputs }
              Out:           BOOLEAN; { Output }
              C:             EVENT;
              State:         States;
BEGIN
  Out := 0;
  State := Low;
  DIV3 := NOT (DIV2);
  REPEAT
              WAIT_FOR_EVENT (C);
              CASE State OF
              Low:     IF En
                              THEN     BEGIN
                                       Out := 1;
                                       State := High1
                                       END;
              High1:   IF DIV3
                              THEN     BEGIN
                                       Out := 1;
                                       State := High2
                                       END
                              ELSE     BEGIN
                                       Out := 0;
                                       State := Low
                                       END;
              High2:   BEGIN
                              Out := 0;
                              State := Low
                       END
              END
  FOREVER
END.
```

a. Draw a state diagram for PAT.

b. Show the output pattern and the period length for the following input conditions:

   En'
   En · DIV2
   En · DIV3

c. Explain the significance and dependencies of the four inputs of the IEC-like device symbol.
   Why are the names DIV2 and DIV3 given?

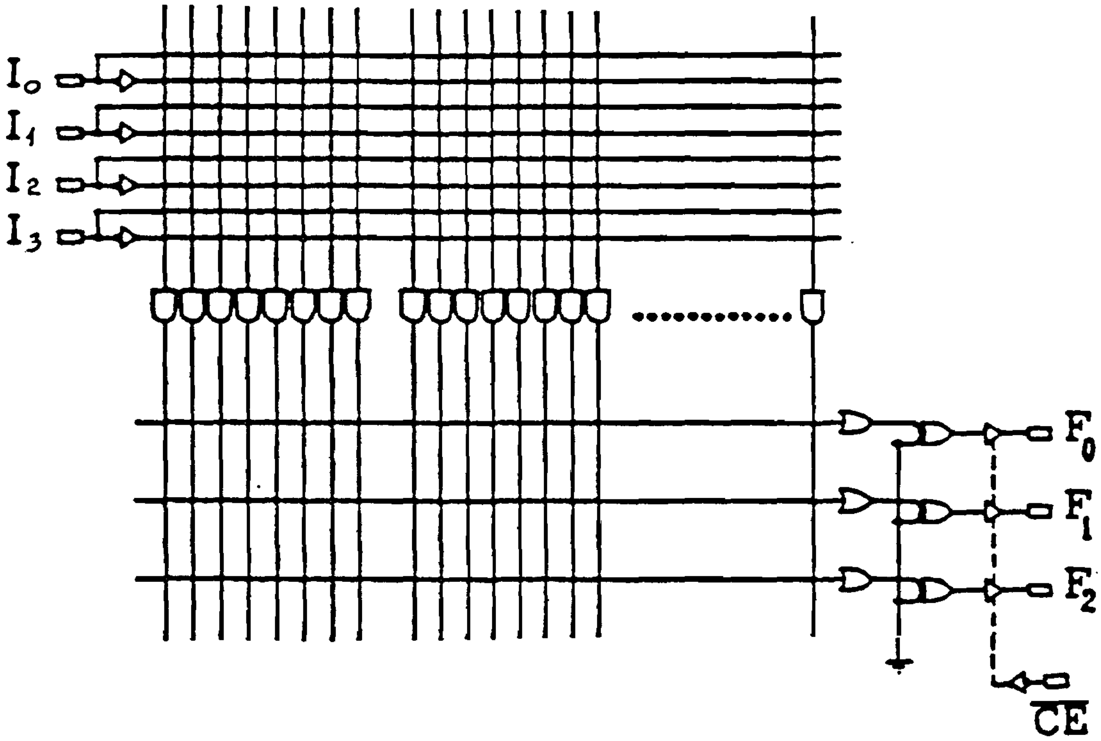
PAT should be realized as a Moore machine.

d. Make a state code assignment "by inspection", minimizing the register size. Explain.

e. Make a coded state table.
   Solve possible state problems arising during power-up and in transient error situations. Explain.

Student name

ANSWER SHEET FOR PROBLEM 3d



Eindhoven International Institute
1992-11-13

EXAMINATION "DESIGN OF DIGITAL SYSTEMS"

Monday 1992-03-09; 8.45 - 11.45 hours

Allowed during examination:
- *"Design of Digital Systems"*
- *English dictionary*
- *Pascal book*

All other material, such as personal notes, solutions to problems etc. is prohibited.

1 Switching functions

Given the following functions of four variables:

$$F(w,x,y,z) = w'y'z + wyz + x'yz' + wx'z$$
$$G(w,x,y,z) = w(y'z' + xy') + x(z' + y \oplus w) + w'yz + x'y'z'$$

Regarding numeric representation of minterms: w has the highest weight and z the lowest one.

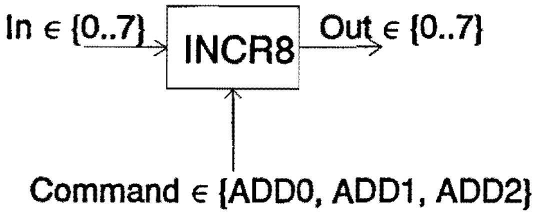
Questions

a. Write function F in numeric form as a sum of minterms; show how the result is obtained.
b. Same for function G.
c. Show that F = G'.
d. Give a function table specifying the combined function F • G.

2 Iterative combinational networks

Consider a system INCR8 with:

inputs:	In	∈ {0..7}	
	Command	∈ {ADD0, ADD1, ADD2}	
output:	Out	∈ {0..7}	
function:	INCR8	Out = In	if Command = ADD0
		Out = (In + 1) MOD 8	if Command = ADD1
		Out = (In + 2) MOD 8	if Command = ADD2



Questions

In your solution, the Command input should never be coded in binary, but given its symbolic values: {ADD0, ADD1, ADD2}.

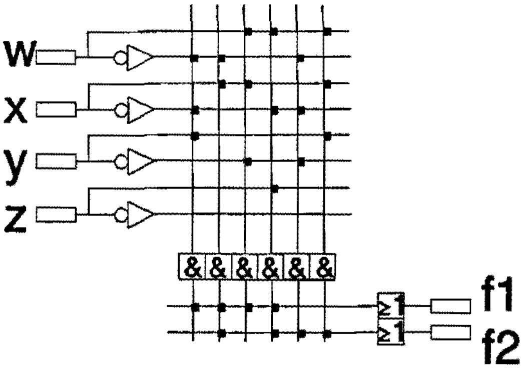
a. Make a behavioural description of INCR8 in Pascal.
b. Specify a truth table of the system (without coding Command!).

In subproblems (c) - (f) below, this system should be realized as a 3-iterative circuit, with 3 identical modules INCR2.

c. Make a block diagram of the 3-iterative circuit, clearly showing all inputs, outputs and inter-module connections, labeled with variable names and their domains/ranges.
d. Make a drawing of a single INCR2 module, labeling inputs and outputs. Give a truth table for INCR2. Specify its operation with a truth table (do not decode the command!)
e. Give a behavioural description in the form of a Pascal function INCR2.
f. Give a behavioural description of the whole 3-iterative network, making use of the INCR2 function declared in question (d).

3 Programmed Logic Array

In the figure below two functions f_1 and f_2 have been realized by means of a PLA with 4 inputs and 2 outputs. Concerning the numeric value of minterms and maxterms, w has the highest weight and z the lowest one.

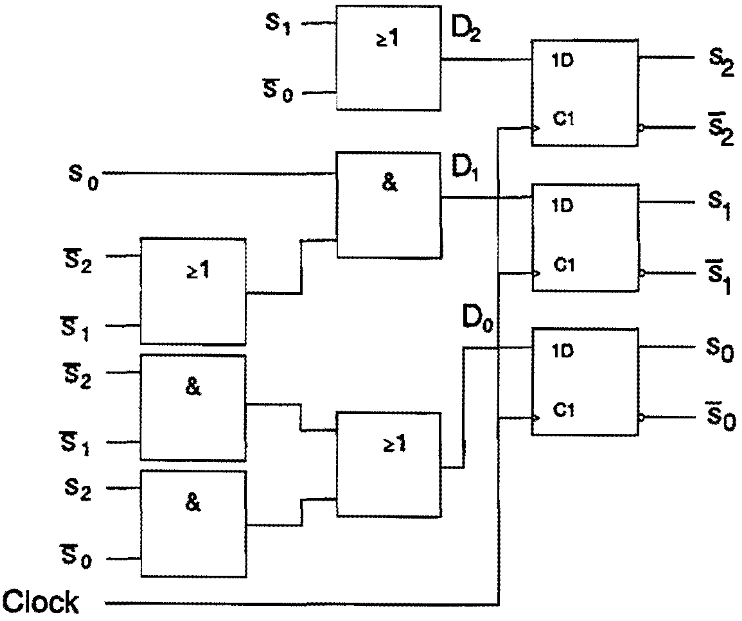


Questions

a. Derive for both functions f_1 and f_2 the sum-of-minterms form, and specify these in numeric format.
b. Same for the product-of-maxterms form for each of the functions.
c. Determine the sum-of-minterm form of the *dual functions* f_{1d} and f_{2d}, and give their numeric representation.
d. Program the PLA in order to realize both functions f_{1d} and f_{2d}.
Use for your solution answer sheet 3d.

4 Finite state machine.

In the figure below a Finite State Machine (FSM) has been drawn.

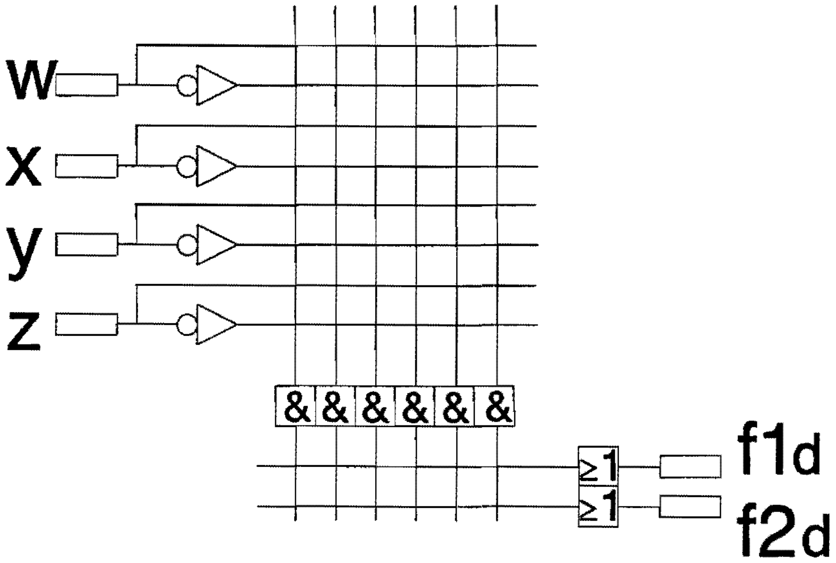


Questions

a. Determine the exitation functions of this machine.
b. Work out a state table.
c. Draw the complete state diagram, specifying all different states and transitions.
(The diagram consists of a number of initial states and a cycle.)

Student name

ANSWER SHEET FOR PROBLEM 3d



Appendix B

Literature

Below, literature can be found related to subjects covered in this course.

General
Lewin D.,
Design of Logic Systems,
Van Nostrand Reinhold, 1985, ISBN 0-442-30606-7.

Dietmeyer D.L.,
Logic Design of Digital Systems, Second Edition,
Allyn and Bacon, 1978, ISBN 0-205-06122-2.

Algorithmic model
Ercegovac M.D. and Lang T.,
Digital Systems and Hardware/Firmware Algorithms,
John Wiley & Sons, 1985, ISBN 0-471-63368-2.

Davio M., Deschamps J.-P. and Thayse A.,
Digital Systems with Algorithmic Implementation,
John Wiley & Sons, 1983, ISBN 0-471-10414-0.

Algorithmic State Machines
Wiatrowski C.A. and House C.H.,
Logic Circuits and Microcomputer Systems,
McGraw-Hill, 1980, ISBN 0-07-066632-6.

Clare C.R.,
Designing Logic Systems Using State Machines,
McGraw-Hill, 1973.

Appendix C

IEC-Symbols

Logic Symbology and dependency notation‡ (Frederic A. Mann†)

From: Digital Hardware Design (John Peatman)

A1-1 INTRODUCTION

The International Electrotechnical Commission (IEC) has been developing a very powerful symbolic language that can show the relationship of each input of a digital logic circuit to each output without showing explicitly the internal logic. At the heart of the system is dependency notation, which will be explained in Sec. A1-4.

The system was introduced in the United States in a rudimentary form in IEEE/ANSI Standard Y32.14-1973. Lacking at that time a complete development of dependency notation, it offered little more than a substitution of rectangular shapes for the familiar distinctive shapes for representing the basic functions of AND, OR, negation, etc. This is no longer the case.

† Manager, Commercial Product Specifications, Semiconductor Group, Texas Instruments Incorporated, MS84, P.O. Box 225012, Dallas, Texas 75265. Member, IEEE Committee SCC 11.9 and IEEE representative to IEC Technical Committee IC-3, Working Group 2.

*‡ Taken by permission from Texas Instruments' **TTL Data Book for Design Engineers**, 3rd edition, now in preparation. To purchase the symbol standards (as they become available), write to the American National Standards Institute, Inc. (1430 Broadway, New York, NY 10018) for IEC Publication 617-12 and to the Institute of Electrical and Electronics Engineers, Inc. (345 East 47th Street, New York, NY 10017) for the revised IEEE Std 91/ANSI Y32.14.*

Internationally, Working Group 2 of IEC Technical Committee TC-3 is preparing a new document (Publication 617-12) that will consolidate the original work started in the mid-1960s and published in 1972 (Publication 117-15) and the amendments and supplements that have followed. Similarly for the United States, IEEE Committee SCC 11.9 is revising the publication IEEE Std 91/ANSI Y32.14. Texas Instruments is participating in the work of both organizations and this third edition of the TTL Data Book introduces new logic symbols in anticipation of the new standards. When changes are made as the standards develop, future editions of this book will take those changes into account. Unfortunately, time and publication schedules have prevented the preparation of symbols for all the devices. This work will continue.

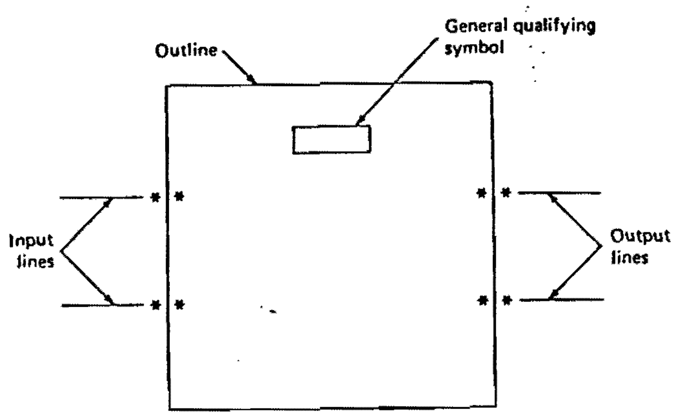
The following explanation of the new symbolic language is necessarily brief and greatly condensed from what the standards publications will finally contain. This is not intended to be sufficient for those people who will be developing symbols for new devices. It is primarily intended to make possible the understanding of the symbols used in this book; comparing the symbols with functional block diagrams and/or function tables will further help that understanding.

A1-2 SYMBOL COMPOSITION

A symbol comprises an outline or a combination of outlines together with one or more qualifying symbols, The shape of the symbols is not significant. As shown in Fig. A1-1, general qualifying symbols are used to tell exactly what logical operation is performed by the elements. Table A1-1 shows the general qualifying symbols used in this data book. Input lines are placed on the left and output lines are placed on the right. When an exception is made to that convention, the direction of signal flow is indicated by an arrow, as shown in Fig. A1-11.

All outputs of an element always have identical internal logic states determined by the function of the element except when otherwise indicated by an associated qualifying symbol inside the element. The outlines of elements may be joined or embedded, in which case the following conventions apply. There is no logic connection between the elements when the line common to their outlines is in the direction of information flow. There is at least one logic connection between the elements when the line common to their outlines is perpendicular to the direction of information flow. The number of logic connections between elements will be clarified by the use of qualifying symbols and this is discussed further under that topic. If no indications are shown on either side of the common line, it is assumed there is only one connection.

When a circuit has one or more inputs that are common to more than one element of the circuit, the common-control block may be used. This is the only distinctively shaped outline used in the IEC system. Fig. A1-2 shows that, unless otherwise qualified by dependency notation, an input to the common-control block may be used. This is the only distinctively shaped outline used in the IEC system. Fig. A1-2 shows that, unless otherwise qualified by dependency notation, an input to the common-control block is an input to each of the elements below the common-control block.



*Possible positions for qualifying symbols relating to inputs and outputs.

Figure A1-1. Symbol composition.

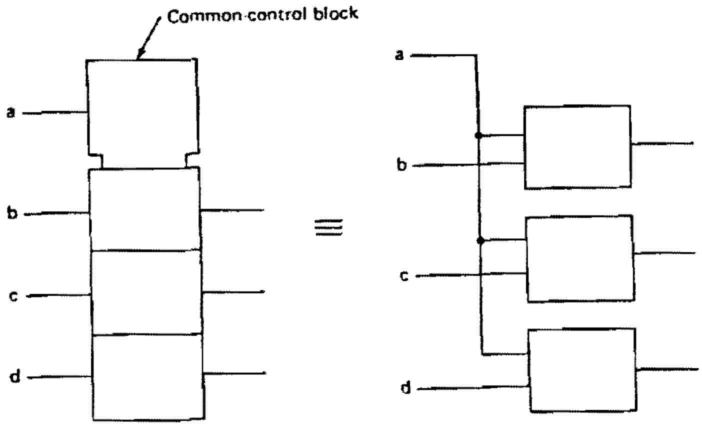


Figure A1-2. Illustration of common-control block.

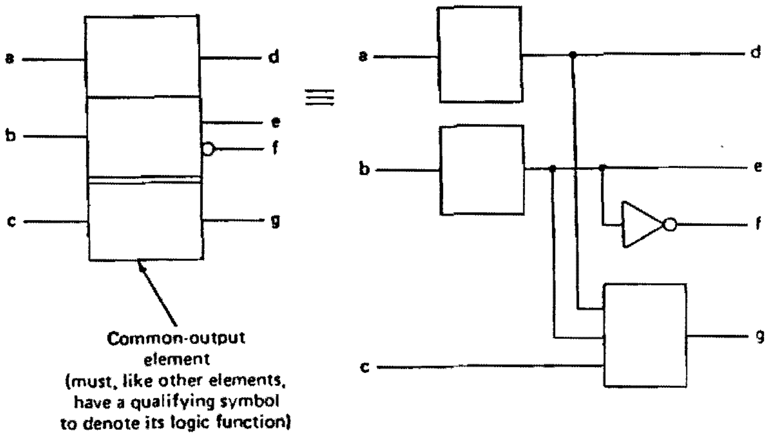

a
b
c
d
e
f
g
≡
Common-output element (must, like other elements, have a qualifying symbol to denote its logic function)

Figure A1-3. Illustration of common-output element.

A common output depending on all elements of the array can be shown as the output of a common-output element. Its distinctive visual feature is the double line at its top. In addition the common-output element may have other inputs as shown in Fig. A1-3. The function of the common-output element must be shown by use of a general qualifying symbol.

A1-3 QUALIFYING SYMBOLS

Table A1-1 shows the general qualifying symbols used in this data book. Qualifying symbols for inputs and outputs are shown in Table A1-2 and will be familiar to most users, with the possible exception of the logic polarity indicators. The older logic negation indicator means that the external 0 state produces the internal 1 state. The internal 1 state means the active state. Logic negation may be used in pure logic diagrams; in order to tie the external 1 and 0 logic states to the levels H (high) and L (low), a statement of whether positive logic (1 = H, 0 = L) or negative logic (1 = L, 0 = H) is being used is required or must be assumed. Logic polarity indicators eliminate the need for calling out the logic convention and are used in this data book in the symbology for actual devices. The presence of the triangular polarity indicator indicates that the L logic level will produce the internal 1 state (the active state) or that, in the case of an output, the internal 1 state will produce the external L level. Note how the active direction of transition for a dynamic input is indicated in positive logic, negative logic, or with polarity indication.

Table A1-1. General Qualifying Symbols

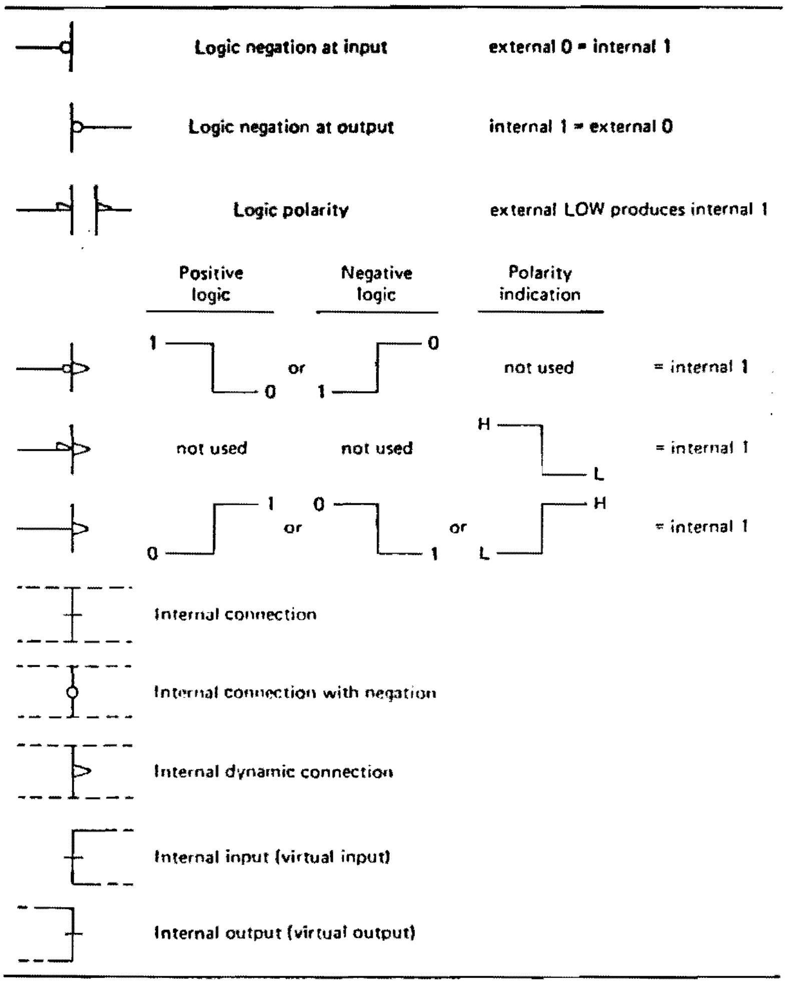
≥1	OR
&	AND
=1	Exclusive-OR
=	All inputs at same state
2k	Even number of inputs active
2k+1	Odd number of inputs active
1	One input active
▷	Buffer, driver, amplifier
⎎	Schmitt trigger
X/Y	Coder, code converter, BCD/DEC, BIN/BCD, etc.
MUX	Multiplexer
DMUX	Demultiplexer
Σ	Adder
P-Q	Subtracter
CPG	Look-ahead carry generator
π	Multiplier
COMP	Magnitude comparator
ALU	Arithmetic logic unit
⎍	Retriggerable monostable
1⎍	Nonretriggerable monostable
G ⎍⎍	Astable element. Showing ⎍⎍ is optional.
!G ⎍⎍	Synchronously starting astable
G! ⎍⎍	Astable element stopping with completed pulse
SRGm	Shift register (m = number of bits)
CTRm	Counter (m = number of bits)
CTRDIVm	Counter with cycle length = m
ROM	Read-only memory
RAM	Random-access memory

The internal connections between logic elements abutted together in a symbol may be indicated by the symbols shown. Each logic connection may be shown by the presence of qualifying symbols at one or both sides of the common line and if confusion can arise about the numbers of connections, use can be made of one of the internal connection symbols.

The internal (virtual) input is an input originating somewhere else in the circuit and is not connected directly to a terminal. The internal (virtual) output is likewise not connected directly to a terminal. The application of internal inputs and outputs requires an understanding of dependency notation, which is explained in Sec. A1-4.

In an array of elements, if the same general qualifying symbol and the same qualifying symbols associated with inputs and outputs would appear inside all the elements of the array, these qualifying symbols are usually shown only in the first element. This is done to reduce clutter and to save time in recognition.

Table A1-2. Qualifying Symbols for Inputs and Outputs

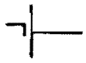
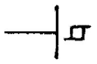
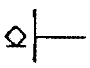
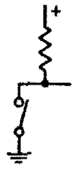
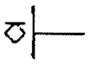
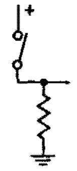
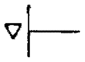
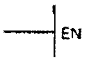
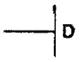
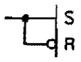
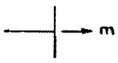
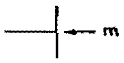
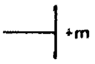
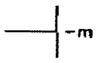
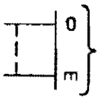
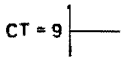
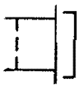
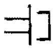
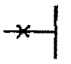
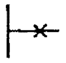


	Logic negation at input		external 0 = internal 1	
	Logic negation at output		internal 1 = external 0	
	Logic polarity		external LOW produces internal 1	
	Positive logic	Negative logic	Polarity indication	
	1 0 or	1 0	not used	= internal 1
	not used	not used	H L	= internal 1
	0 1 or	0 1 or	L H	= internal 1
	Internal connection			
	Internal connection with negation			
	Internal dynamic connection			
	Internal input (virtual input)			
	Internal output (virtual output)			

A1-3.1 Symbols Inside the Outline

Table A1-3 shows some symbols used inside the outline. Note particularly that open-collector, open-emitter, and three-state outputs have distinctive symbols. Also note that an EN input affects all of the outputs of the circuit and has no effect on inputs. When an enable input affects only certain outputs and/or does affect one or more inputs, a form of dependency notation will indicate this. The effects of the EN input on the various types of outputs are shown.

Table A1-3. Symbols Inside the Outline

	Postponed output (of a pulse-triggered flip-flop). The output changes when input initiating change (e.g., a C input) returns to its initial external state or level.
	Bithreshold input (input with hysteresis)
	NPN open-collector or similar output that can supply a relatively low-impedance L level when not turned off. Requires external pull-up. Capable of positive-logic wired-AND connection. 
	NPN open-emitter or similar output that can supply a relatively low-impedance H level when not turned off. Requires external pull-down. Capable of positive-logic wired-OR connection. 
	3-state output
	Enable input When at its internal 1-state, all outputs are enabled. When at its internal 0-state, ◇outputs are off, ▽ outputs are at normally defined internal logic states and at external high-impedance state, and all other outputs (e.g., totem-poles) are at the internal 0-state.
J, K, R, S, T	Usual meanings associated with flip-flops.
	Data input to a storage element equivalent to: 
 	Shift right (left) inputs m = 1, 2, 3, etc.
 	Counting up (down) inputs m = 1, 2, 3, etc.
	Binary grouping. m is the highest power of 2.
	Content equals (e.g., 9)
	Input line grouping . . . . indicates 2 or more terminals used to implement a single logic input, e.g., a differential input: 
 	Nonlogic input (output)

It is particularly important to note that a D input is always the data input of a storage element. At its internal 1 state, the D input sets the storage element to its 1 state, and at its internal 0 state it resets the storage element to its 0 state.

The binary grouping symbol is important. Binary-weighted inputs are arranged in order and the binary weights of the least-significant and the most-significant lines are indicated by numbers. In this data book weights of input and output lines will be represented by powers of 2 only when the binary grouping symbol is used; otherwise, decimal numbers will be used. The grouped inputs generate an internal number on which a mathematical function can be performed or that can be an identifying number for dependency notation. See Fig. A1-24. A frequent use is in addresses for memories.

Reversed in direction, the binary grouping symbol can be used with outputs. The concept is analogous to that for the inputs, and the weighted outputs will indicate the internal number assumed to be developed by the circuit.

Other symbols are used inside the outlines in this catalog in accordance with the IEC/IEEE standard but are not shown here. Generally these are associated with arithmetic operations and are self-explanatory.

When nonstandardized information is shown inside an outline, it is usually enclosed in square brackets [like these].

A1-4 DEPENDENCY NOTATION

A1-4.1 General Explanation

Dependency notation is the powerful tool that sets the IEC symbols apart from previous systems and makes compact, meaningful symbols possible. It provides the means of denoting the relationship between inputs, outputs, or inputs and outputs without actually showing all the elements and interconnections involved. The information provided by dependency notation supplements that provided by the qualifying symbols for an element's function.

In the convention for the dependency notation, use will be made of the terms "affecting" and "affected". In the case where it is not evident which inputs must be considered as being the affecting or the affected ones (e.g., if they stand in and AND relationship), the choice may be made in any convenient way.

So far, ten types of dependency have been defined and all of these are used in this data book. They are listed below in the order in which they are presented and are summarized in Table A1-4 in Sec. A1-4.11.

Section	Dependency type or other subject
A1-4.2	G, AND
A1-4.3	General rules for dependency notation
A1-4.4	V, OR
A1-4.5	N, negate, exclusive-OR
A1-4.6	Z, interconnection
A1-4.7	C, control
A1-4.8	S, set and R, reset
A1-4.9	EN, enable
A1-4.10	M, mode
A1-4.11	A, address
A1-4.12	Use of a coder to produce affecting inputs
A1-4.13	Use of binary grouping to produce affecting inputs
A1-4.14	Sequence of input labels
A1-4.15	Sequence of output labels

A1-4.2 G (AND) Dependency

A common relationship between two signals is to have then ANDed together. This has traditionally been shown by explicitly drawing an AND gate with the signals connected to the inputs of the gate. The 1972 IEC publication and the 1973 IEEE/ANSI standard showed several ways to show this AND relationship using dependency notation. While nine other forms of dependency have since been defined, the ways to invoke AND dependency are now reduced to one.

In Fig. A1-4 input *b* is ANDed with input *a* and the complement of *b* is ANDed with *c*. The letter G has been chosen to indicate AND relationships and is placed at input *b*, inside the symbol. An arbitrary number (1 has been used here) is placed after the letter G and also at each affected input. Note the bar over the 1 at input c.

In Fig. A1-5, output *b* affects input a with an AND relationship. The lower example shows that it is the internal logic state of *b*, unaffected by the negation sign, that is ANDed. Fig. A1-6 shows input *a* to be ANDed with a dynamic input *b*.

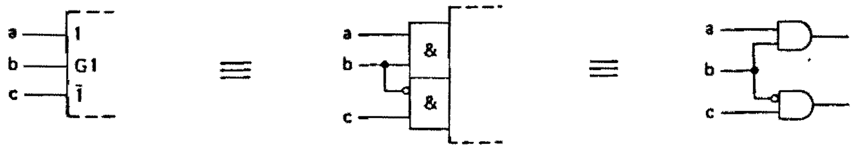


Figure A1-4. G dependency between inputs.

The rules for G-dependency can be summarized thus: When a G*m* input or output (*m* is a number) stands at its internal 1 state, all inputs and outputs affected by G*m* stand at their normally defined internal logic states. When the G*m* input or output stands at its 0 state, all inputs and outputs affected by Gm stand at their internal 0 states.

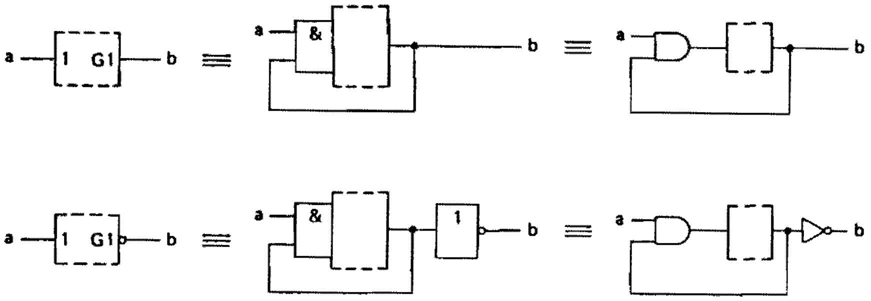


Figure A1-5. G dependency between outputs and inputs.

A1-4.3 Conventions for the Application of Dependency Notation in General

The rules for applying dependency relationships in general follow the same pattern as was illustrated for G-dependency.

Application of dependency notation is accomplished by:

1 Labeling the input (or output) affecting other inputs or outputs with a letter symbol indicating the relationship involved (e.g., G for AND) followed by an identifying number, arbitrarily chosen.

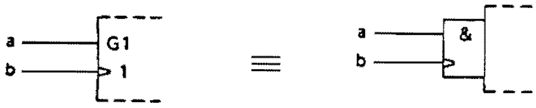
2 Labeling each input or output affected by that affecting input (or output) with that same number.

If it is the complement of the internal logic state of the affecting input or output that does the affecting, then a bar is placed over the identifying numbers at the affected inputs or outputs. See Fig. A1-4.

If the affected input or output requires a label to denote its function (e.g., D), this label will be prefixed by the identifying number of the affecting input. See Fig. A1-12.

If an input or output is affected by more than one affecting input, the identifying numbers of each of the affecting inputs will appear in the label of the affected one, separated by commas.

Figure A1-6. G dependency with a dynamic input.



The left-to-right sequence of these numbers is the same as the sequence of the affecting relationships. See Fig. A1-12.

If the labels denoting the functions of affected inputs or outputs must be numbers, the identifying numbers to be associated with both affecting inputs and affecting inputs or outputs will be replaced by another character selected to avoid ambiguity (e.g., Greek letters). See Fig. A1-8.

Figure A1-7. OR'ed affecting inputs.

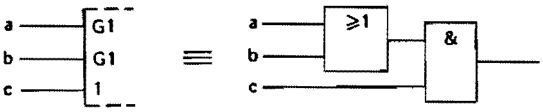
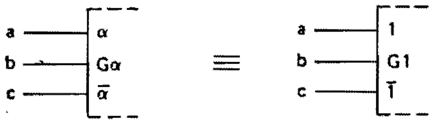


Figure A1-8. Substitution for numbers.



A1-4.4 V (OR) Dependency

The symbol denoting OR-dependency is the letter V. See Fig. A1-9.

When a V*m* input or output stands at its internal 1 state, all inputs and outputs affected by V*m* stand at their internal 1 states. When the V*m* input or output stands at its internal 0 state, all inputs and outputs affected by V*m* stand at their normally defined internal logic states.

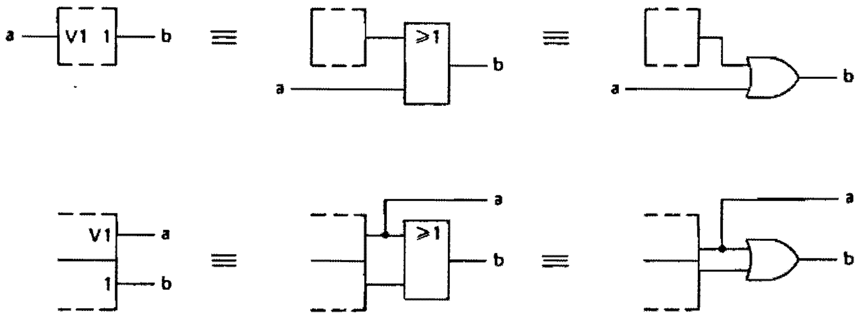
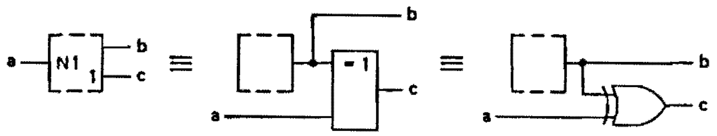


Figure A1-9. V(OR) dependency

A1-4.5 N (Negate) (X-OR) Dependency

The symbol denoting negate dependency is the letter N. See Fig. A1-10. Each input or output affected by an N*m* input or output stands in an exclusive-OR relationship with the N*m* input or output.



If a = 0, c = b
If a = 1, c = $\bar{b}$

Figure A1-10. N (Negate) (X-OR) dependency.

When an N*m* input or output stands at its internal 1 state, the internal logic state of each input and each output affected by N*m* is the complement of what it would otherwise be. When an N*m* input or output stands at its internal 0 state, all inputs and outputs affected by N*m* stand at their normally defined internal logic states.

A1-4.6 Z (Interconnection) Dependency

The symbol denoting interconnection dependency is the letter Z.

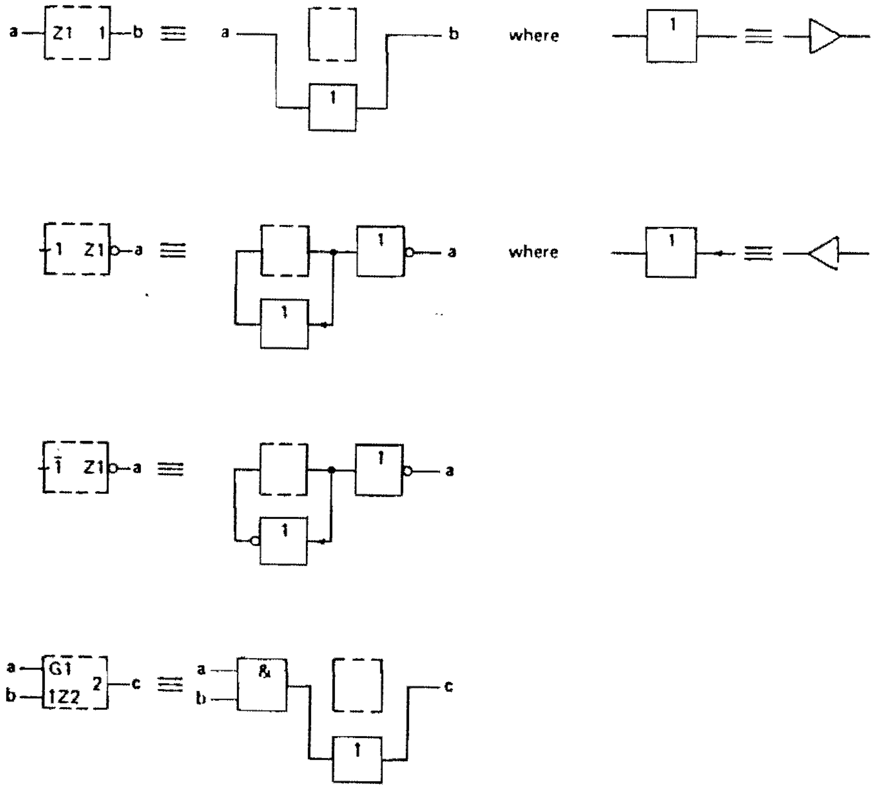


Figure A1-11. Z (interconnection) dependency.

Interconnection dependency is used to indicate the existence of internal logic connections between inputs, outputs, internal inputs, and/or internal outputs.

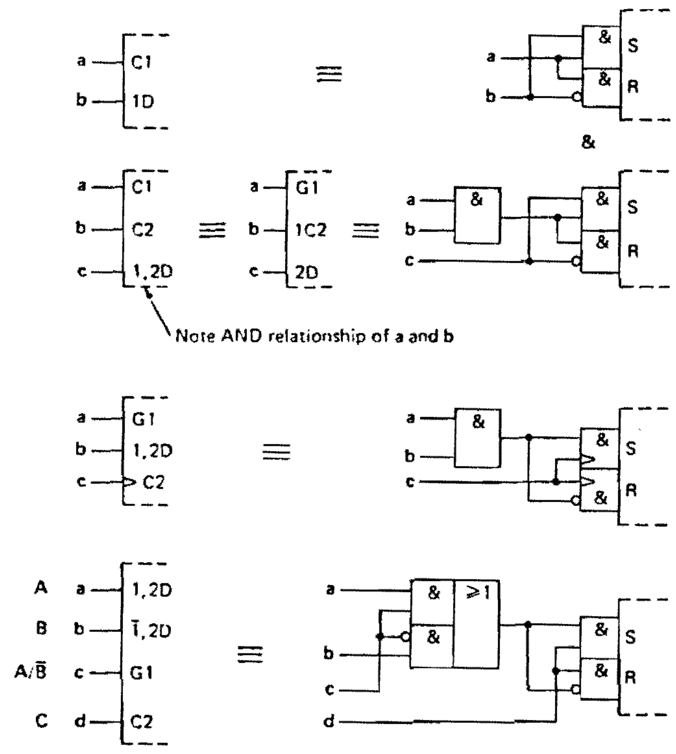
The internal logic state of an input or output affected by a *Zm* input or output will be the same as the internal logic state of the *Zm* input or output. See Fig. A1-11.

A1-4.7 C (Control) Dependency

The symbol denoting control dependency is the letter C.

Control inputs are usually used to enable of disable the D (data) inputs of storage elements. They may take on their internal 1 states (be active) either statically or dynamically. In the latter case the dynamic input symbol is used as shown in the third example of Fig. A1-12.

When a C*m* input or output stands at its internal 1 state, the inputs affected by C*m* have their normally defined effect on the function of the element (i.e., these inputs are enabled). When a C*m* input or output stands at its internal 0 state, the inputs affected by C*m* are disabled and have no effect on the function of the element.



A/B̄ selects which of A or B is stored when C goes low.

Figure A1-12. C (control) dependency.

A1-4.8 S (Set) and R (Reset) Dependencies

The symbol denoting set dependency is the letter S. The symbol denoting reset dependency is the letter R. Set and reset dependencies are used if it is necessary to specify the effect of the combination R = S = 1 on a bistable element. Case 1 in Fig. A1-13 does not use S or R dependency.

When an S*m* input is at its internal 1 state, outputs affected by the S*m* input will react, regardless of the state of an R input, as they normally would react to the combination S = 1, R = 0. See cases 2, 4, and 5 in Fig. A1-13.

When an R*m* input is at its internal 1 state, outputs affected by the R*m* input will react, regardless of the state of an S input, as they normally would react to the combination S = 0, R = 1. See cases 3, 4 and 5 in Fig. A1-13.

When an S*m* or R*m* input is at its internal 0 state, it has no effect. Note that the noncomplementary output patterns in cases 4 and 5 are only pseudo stable. The simultaneous return of the inputs to S = R = 0 produces an unforeseeable stable and complementary output pattern.

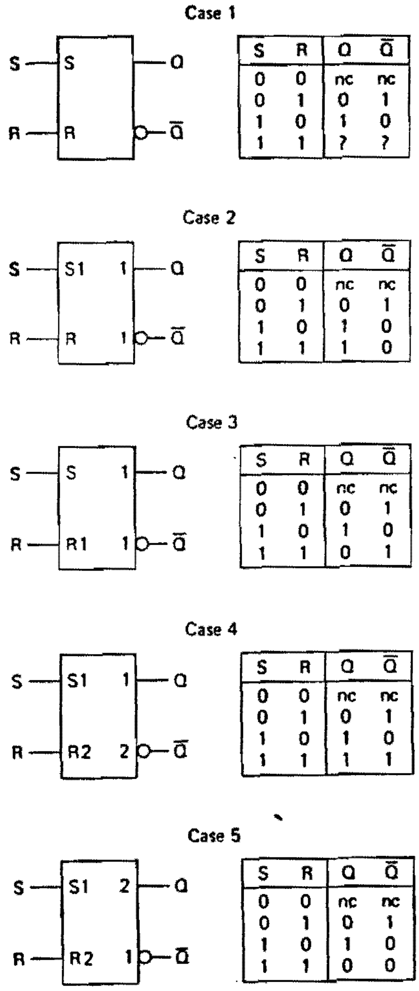
A1-4.9 EN (Enable) Dependency

The symbol denoting enable dependency is the combination of letters EN.

An EN*m* input has the same effect on outputs as an EN input (see Sec. A1-3.1), but it can affect less than all of the outputs. It can also affect inputs. By contrast, an EN input affects all outputs and no inputs. The effect of an EN*m* input on an affected input is identical to that of a C*m* input. See Fig. A1-14.

When an EN*m* input stands at its internal 1 state, the inputs affected by EN*m* have their normally defined effect on the function of the element and the outputs affected by this input stand at their normally defined internal logic states, i.e., these inputs and outputs are enabled.

When an EN*m* input stands at its internal 0 state, the inputs affected by EN*m* are disabled and have no effect on the function of the element, and the outputs affected by EN*m* are also disabled. Open-collector outputs are turned off, three-state outputs stand at their normally defined internal logic states but externally exhibit high impedance, and all other outputs (e.g., totem-pole outputs) stand at their internal 0 states.



Case 1

S	R	Q	$\overline{Q}$
0	0	nc	nc
0	1	0	1
1	0	1	0
1	1	?	?

Case 2

S	R	Q	$\overline{Q}$
0	0	nc	nc
0	1	0	1
1	0	1	0
1	1	1	0

Case 3

S	R	Q	$\overline{Q}$
0	0	nc	nc
0	1	0	1
1	0	1	0
1	1	0	1

Case 4

S	R	Q	$\overline{Q}$
0	0	nc	nc
0	1	0	1
1	0	1	0
1	1	1	1

Case 5

S	R	Q	$\overline{Q}$
0	0	nc	nc
0	1	0	1
1	0	1	0
1	1	0	0

0 = external 0 state
1 = external 1 state
nc = no change
? = unspecified

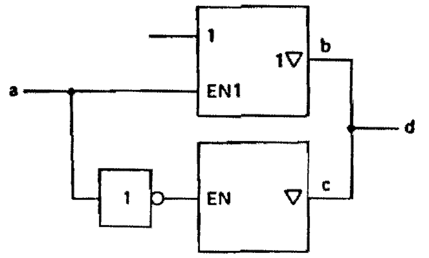
Figure A1-13. S (set) and R (reset) dependencies.

A1-4.10 M (Mode) Dependency

The symbol denoting mode dependency is the letter M.

Mode dependency is used to indicate that the effects of particular inputs and outputs of an element depend on the mode in which the element is operating.

If an input or output has the same effect in different modes of operation, the identifying numbers of the relevant affecting M*m* inputs will appear in the label of that affected input or output between parentheses and separated by commas. See Fig. A1-19.

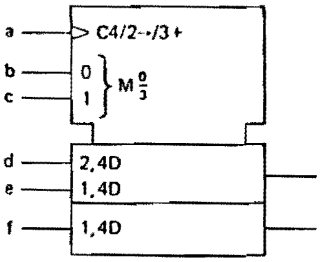


If a = 0, b disabled and d = c
If a = 1, c disabled and d = b

Figure A1-14. EN (enable) dependency.

M Dependency Affecting Inputs. M dependency affects inputs the same as C dependency. When an M*m* input or M*m* output stands at its internal 1 state, the inputs affected by this M*m* input or M*m* output have no effect on the function of the element. When an affected input has several sets of labels separated by slashes, any set in which the identifying number of the M*m* input or M*m* output appears has no effect and is to be ignored. This represents disabling of some of the functions of a multifunction input.

The circuit in Fig. A1-15 has two inputs, *b* and *c*, that control which one of four modes (0,1,2 or 3) will exist at any time. Inputs *d*,*e*, and *f* are D inputs subject to dynamic control (clocking) by the *a* input. The numbers 1 and 2 are in the series chosen to indicate the modes so inputs *e* and *f* are only enabled in mode 1 (for parallel loading) and input *d* is only enabled in mode 2 (for serial loading). Note that input *a* has three functions. It is the clock for entering data. In mode 2, it causes right shifting of data, which means a shift away from the control block. In mode 3, it causes the contents of the register to be incremented by one count.



Note that all operations are synchronous.

In mode 0 (b = 0, c = 0), the outputs remain at their existing states as none of the inputs has an effect.

In mode 1 (b = 1, c = 0), parallel/loading takes place through inputs e and f.

In mode 2 (b = 0, c = 1), shifting down and serial loading through input d take place.

In mode 3 (b = c = 1), counting up by increment of 1 per clock pulse take place.

Figure A1-15. M (mode) dependency affecting inputs.

M (Mode) Dependency Affecting Outputs. When an M*m* input or M*m* output stands at its internal 1 state, the affected outputs stand at their normally defined internal logic states, i.e., the outputs are enabled.

When an M*m* input or M*m* output stands at its internal 0 state, at each affected output any set of labels containing the identifying number of that M*m* input or M*m* output has no effect and is to be ignored. When an output has several different sets of labels separated by slashes (e.g., C4/2→/3+), only those sets in which the identifying number of this M*m* output appears are to be ignored.

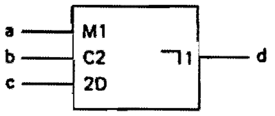


Figure A1-16. Type of flip-flop determined by mode.

In Fig. A1-16, mode 1 exists when the *a* input stands at its internal 1 state. The delayed output symbol is effective only in mode 1 (when input *a* = 1) in which case the device functions as a pulse-triggered flip-flop. See Sec. A1-5. When input *a* = 0, the device is not in mode 1 so the delayed output symbol has no effect and the device functions as a transparent latch.

In Fig. A1-17, if input *a* stands at its internal 1 state establishing mode 1, output *b* will stand at its internal 1 state when the content of the register equals 9. Since output *b* is located in the common-control block with no defined function outside of mode 1, this output will stand at its internal 0 state when input *a* stands at its internal 0 state, regardless of the register content.

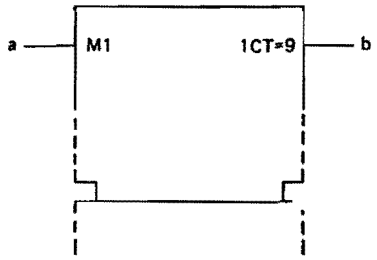


Figure A1-17. Disabling an output of the common-control block.

In Fig. A1-18, if input *a* stands at its internal 1 state establishing mode 1, output *b* will stand at its internal 1 state when the content of the register equals 15. If input *a* stands at its internal 0 state, output *b* will stand at its internal 1 state when the content of the register equals 0.

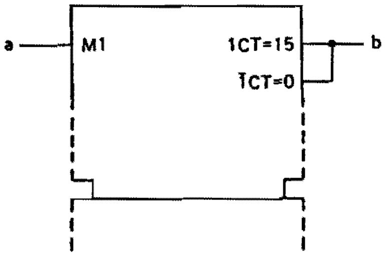


Figure A1-18. Determining an output's function.

In Fig. A1-19 inputs *a* and *b* are binary weighted to generate the numbers 0,1,2, or 3. This determines which one of the four modes exists.

At output *e* the label set causing negation (if *c* = 1) is effective only in modes 2 and 3. In modes 0 and 1 this output stands at its normally defined state as if it had no labels.

At output *f* the label set has effect when the mode is not 0 so output e is negated (if c = 1) in modes 1,2, and 3. In mode 0 the label set has no effect so the output stands at its normally defined state. In this example, $\bar{0}$,4 is equivalent to (1/2/3)4.

At output *g* there are two label sets. The first set, causing negation (if *c* = 1), is effective only in mode 2. The second set, subjecting *g* to AND dependency on *d*, has effect only in mode 3.

Note that in mode 0 none of the dependency relationships have any effect on the outputs, so e,*f*, and *g* will all stand at the same state.

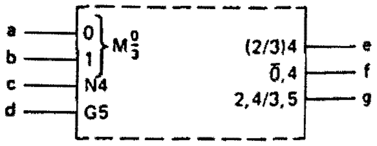


Figure A1-19. Dependent relationships affected by mode.

A1-4.11 A (Address) Dependency

The symbol denoting address dependency is the letter A.

Address dependency is used to obtain a clear representation of those elements, particularly memories, that use address control inputs to select specified sections of a multidimensional array. Such a section of a memory array is usually called a word. The purpose of address dependency is to allow a symbolic presentation of only a single general case of the sections of the array, rather than requiring a symbolic presentation of the entire array. An input of the array shown at a particular element of this general section is common to the corresponding elements of all selected sections of the array. An output of the array shown at a particular element of this general section is the result of the OR function of the outputs of the corresponding elements of selected sections. If the label of an output of the array shown at a particular element of this general section indicates that this output is an open-circuit output or a three-state output, then this indication refers to the output of the array and not to those of the sections of the array.

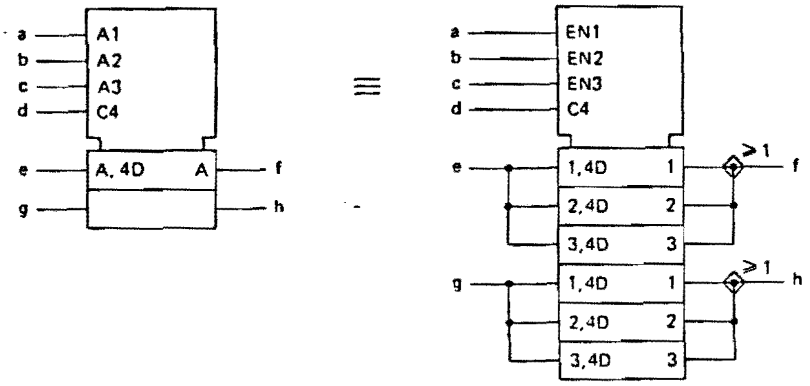


Figure A1-20. *A* (address) dependency.

Inputs that are not affected by any affecting address input have their normally defined effect on all sections of the array, whereas inputs affected by an address input have their normally defined effect only on the section selected by that address input.

An affecting address input is labeled with the letter A followed by an identifying number that corresponds with the address of the particular section of the array selected by this input.

Within the general section presented by the symbol, inputs and outputs affected by an A*m* input are labeled with the letter A, which stands for the identifying numbers, i.e., the addresses, of the particular sections.

Fig. A1-20 shows a 3-word by 2-bit memory having a separate address line for each word and uses EN dependency to explain the operation. To select word 1, input *a* is taken to its 1 state, which establishes mode 1. Data can now be clocked into the inputs marked 1,4D. Unless words 2 and 3 are also selected, data cannot be clocked in at the inputs marked 2,4D and 3,4D. The outputs will be the OR functions of the selected outputs, i.e., only those enabled by the active EN functions.

The identifying numbers of affecting address inputs correspond with the addresses of the sections selected by these inputs. They need not necessarily differ from those of other affecting dependency-inputs (e.g., G,V,N,...), because in the general section presented by the symbol they are replaced by the letter A.

If there are several sets of affecting A inputs for the purpose of independent and possibly simultaneous access to sections of the array, then the letter A is modified to 1A,2A.... Because they have access to the same sections of the array, these sets of A inputs may have the same identifying numbers. Fig. A1-21 is another illustration of the concept. Table A1-4 summarizes the dependency notation.

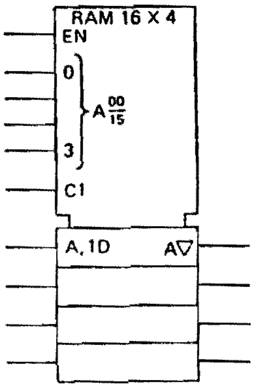


Figure A1-21. Array of 16 sections of four transparent latches with three-state outputs. This comprises a 16-word × 4-bit random-access memory.

A1-4.12 Use of a Coder to Produce Affecting Inputs

It often occurs that a set of affecting inputs is produced by decoding the signals on certain inputs to an element. In such a case one can use the symbol for a coder as an embedded symbol. See Fig. A1-22.

Figure A1-22. Producing various types of dependencies.

If all affecting inputs produced by a coder are of the same type and their identifying numbers correspond with the numbers shown at the outputs of the coder, Y (in the qualifying symbol X/Y) may be replaced by the letter denoting the type of dependency. The indications of the affecting inputs should then be omitted. See Fig. A1-23.

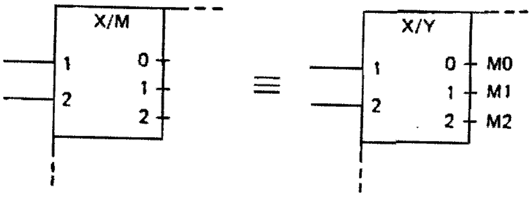


Figure A1-23. Producing one type of dependency.

Table A1-4. Summary of Dependency Notation

Type of dependency	Letter symbol*	Affecting input at its 1 state	Affecting input at its 0 state
Address	A	Permits action (address selected)	Prevents action (address not selected)
Control	C	Permits action	Prevents action
Enable	EN	Permits action	Prevents action of inputs ◇ Outputs off ▽ Outputs at external high impedance, no change in internal logic state Other outputs at internal 0 state
AND	G	Permits action	Imposes 0 state
Mode	M	Permits action (mode selected)	Prevents action (mode not selected)
Negate (X-OR)	N	Complements state	No effect
Reset	R	Affected output reacts as it would to S = 0, R = 1	No effect
Set	S	Affected output reacts as it would to S = 1, R = 0	No effect
OR	V	Imposes 1 state	Permits action
Interconnection	Z	Imposes 1 state	Imposes 0 state

* These letter symbols appear at the *affecting* input (or output) and are followed by a number. Each input (or output) *affected* by that input is labeled with that same number. When the labels EN, R, and S appear at inputs without the following numbers, the descriptions above do not apply. The action of these inputs is described under "Symbols Inside the Outline." See Table A1-3.

A1-4.13 Use of Binary Grouping to Produce Affecting Inputs

If all affecting inputs produced by a coder are of the same type and have consecutive identifying numbers not necessarily corresponding with the numbers that would have been shown at the outputs of coder, use can be made of the binary grouping symbol (Table A1-3). It is followed by the letter denoting the type of dependency followed by

$\frac{m1}{m2}$ The m1 is to be replaced by the smallest identifying number and the m2 by the largest one, as shown in Fig. A1-24.

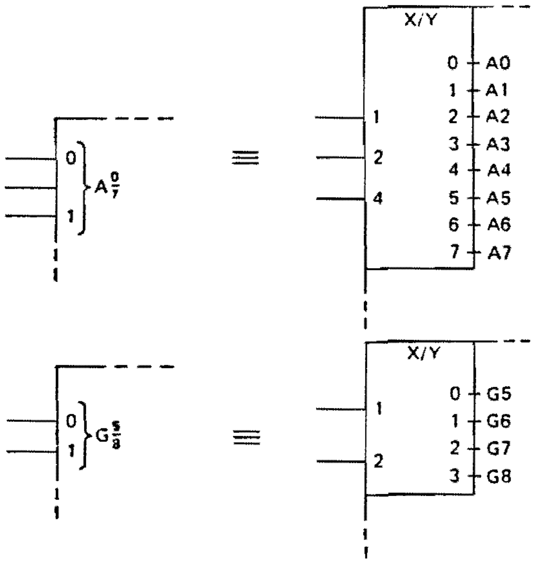


Figure A1-24. Use of the binary grouping symbol.

If an output needs several different sets of labels that represent alternative functions (e.g., depending on the mode of action), these sets may be shown on different output lines that must be connected outside the outline. See Fig. A1-27. However, there are cases in which this method of presentation is not advantageous. In those cases the output may be shown once with the different sets of labels separated by slashes.

Two adjacent identifying numbers of affecting inputs in a set of labels that are not already separated by a nonnumeric character should be separated by a comma.

If a set of labels of an output not containing a slash contains the identifying number of an affecting M*m* input standing at its internal 0 state, this set of labels has no effect on that output.

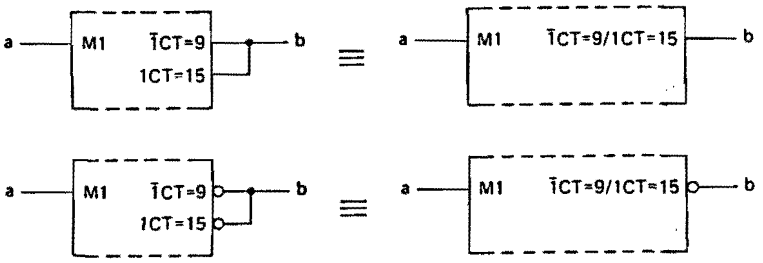


Figure A1-27. Output labels.

A1-5 BISTABLE ELEMENTS

The dynamic input symbol, the postponed output symbol, and dependency notation provide the tools to differentiate four main types of bistable elements and make synchronous and asynchronous inputs easily recognizable. See Fig. A1-28. The first column shows the essential distinguishing features; the other columns show examples.

Transparent latches have a level-operated control input. The D input is active as long as the C input is at its internal 1 state. The outputs respond immediately. Edge-triggered elements accept data from D,J,K,R, or S inputs on the active transition of C. Pulse-triggered elements require the setup of data before the start of the control pulse; the C input is considered static since the data must be maintained as long as C is at its 1 state. The output is postponed until C returns to its 0 state. The data-lockout element is similar to the pulse-triggered version except that the C input is considered dynamic in that shortly after C goes through its active transition, the data inputs are disabled and data does not have to be held. However, the output is still postponed until the C input returns to its initial external level.

Notice that synchronous inputs can be readily recognized by their dependency labels (1D,1J,1K,1S,1R) compared to the asynchronous inputs (S,R), which are not dependent on the C inputs.

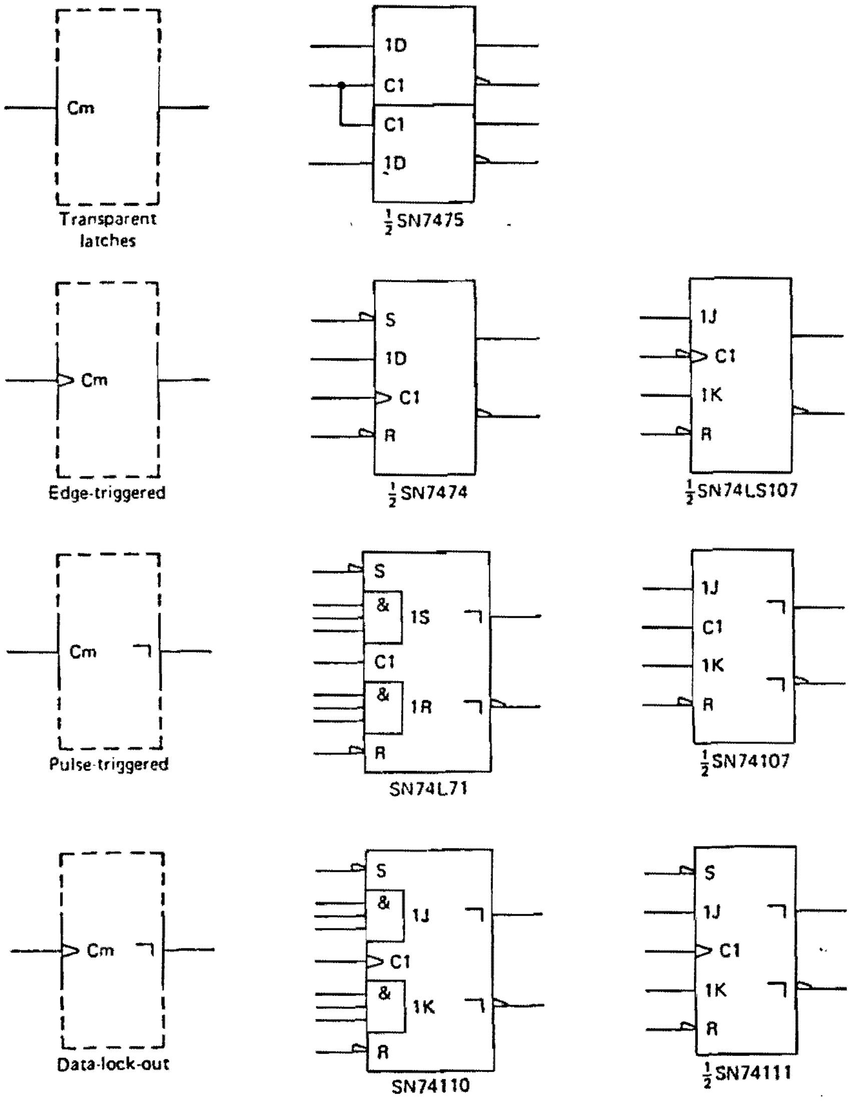


Figure A1-28. Four types of bistable circuits.

Eindhoven International Institute

Course In Electronic Engineering

Survey Nr 276/2	Solutions to exercises of : **INTRODUCTORY SEMESTER 1993** **- Electronic Engineering -**
Subject	: DESIGN OF DIGITAL SYSTEMS
Lecturer	: Ir M.J.M. van WEERT
Author	: Ir M.J.M. van WEERT
Copy	:

Table of Contents

In the appendices, solutions are given to the exercises in the DESIGN OF DIGITAL SYSTEMS SURVEY.

Appendix A

Solutions to Examinations

TECHNISCHE UNIVERSITEIT EINDHOVEN

TENTAMEN DIGITALE SYSTEMEN II
5A010

TOTAL: 8 sheets
of which 4 answersheets

Saturday 10 november 1990
time: 09.00 - 12.00 hours

ANSWERINGSHEET 1

ANSWERSHEETS
NAME:____________________________________

I.D. NUMBER:__________________________

1. a) $(a+b)(a+c)(a+d)$

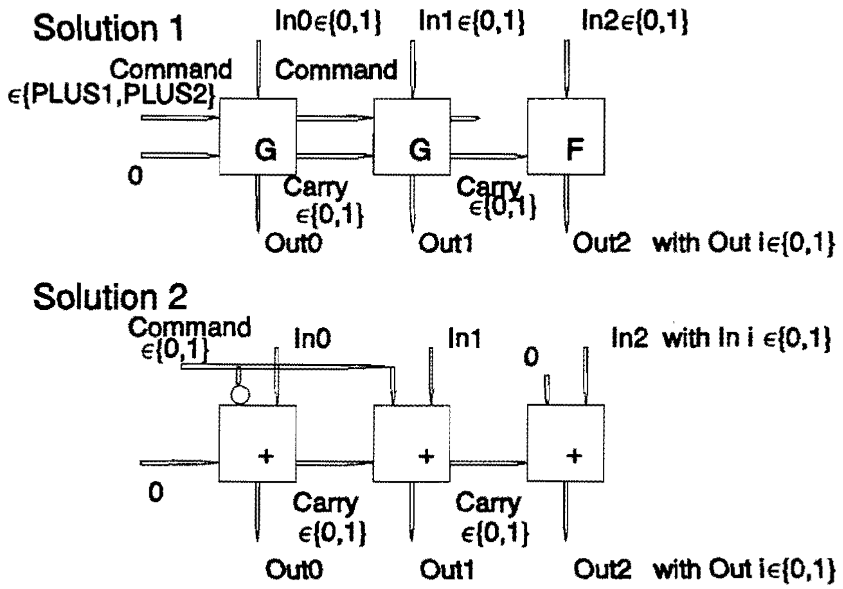
   b) $ab+b'c'$

   c) $(a'+b')c$

   d) $abc'+ab'c$

   e) $a'b'+ab$

2. a) space for drawing the schematic diagram



TENTAMEN DIGITALE SYSTEMEN II
5A010

TOTAL: 8 s.
of which 4 answershee

Saturday 10 november 1990
time: 09.00 - 12.00 hours

ANSWERINGSHEET 2

ANSWERSHEETS
NAME:________________________________

I.D. NUMBER:__________________________

Answer 2 continued

b)

Command	*In i*	*Carry*	*Out i*	*Command*	*Carry*
PLUS 1	**0**	**0**	**1**	**PLUS 2**	**0**
PLUS 1	**1**	**0**	**0**	**PLUS 2**	**1**
PLUS 1	**0**	**1**	**0**	**PLUS 2**	**1**
PLUS 1	**1**	**1**	**1**	**PLUS 2**	**1**
PLUS 2	**0**	**0**	**0**	**PLUS 1**	**0**
PLUS 2	**1**	**0**	**1**	**PLUS 1**	**0**
PLUS 2	**0**	**1**	**1**	**PLUS 1**	**0**
PLUS 2	**1**	**1**	**0**	**PLUS 1**	**1**

3. a) $f_1 = \Sigma(2,3,4,5,6,7,9,11,12,13)$

$f_2 = \Sigma(0,1,4,5,6,7,9,11,14,15)$

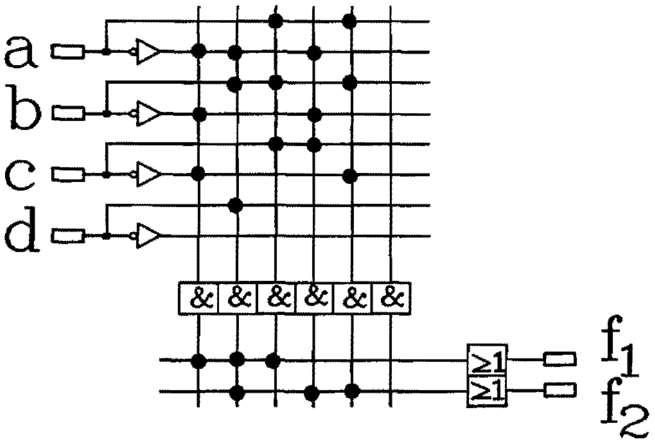
b) $f_1 = \Pi(0,1,8,10,14,15)$

$f_2 = \Pi(2,3,8,10,12,13)$

c) $f_{1d} = \Sigma(0,1,5,7,14,15)$

$f_{2d} = \Sigma(2,3,5,7,12,13)$

d)



TECHNISCHE UNIVERSITEIT EINDHOVEN

TENTAMEN DIGITALE SYSTEMEN II
5A010

TOTAL: 8 sheets
of which 4 answersheets

Saturday 10 november 1990
time: 09.00 - 12.00 hours

ANSWERINGSHEET 3

ANSWERSHEETS
NAME:____

I.D. NUMBER:________________________________

4. For the most significant state bit must hold:

$J = \Sigma 3,\underline{4},\underline{5},(\underline{6},\underline{7}),11,\underline{12},\underline{13},(\underline{14},\underline{15})$

$K = \Sigma \underline{0},\underline{1},\underline{2},\underline{3},4,5,(\underline{6},\underline{7}),\underline{8},\underline{9},\underline{10},\underline{11},13,(\underline{14},\underline{15})$

For the next state bit must hold:

$J = \Sigma 1,\underline{2},\underline{3},(\underline{6},\underline{7}),9,\underline{10},\underline{11},(\underline{14},\underline{15})$

$K = \Sigma \underline{0},\underline{1},3,\underline{4},\underline{5},(\underline{6},\underline{7}),\underline{8},\underline{9},11,\underline{12},\underline{13},(\underline{14},\underline{15})$

And for the least significant state bit must hold:

$J = \Sigma 0,\underline{1},2,\underline{3},\underline{5},(\underline{6},\underline{7}),8,\underline{9},10,\underline{11},12,\underline{13},(\underline{14},\underline{15})$

$K = \Sigma \underline{0},1,\underline{2},3,\underline{4},(\underline{6},\underline{7}),\underline{8},9,\underline{10},11,\underline{12},13,(\underline{14},\underline{15})$

5. a)

current state	next state
c b a	c b a
0 0 0	*0 0 1*
0 0 1	*0 1 0*
0 1 0	*1 0 0*
0 1 1	*0 1 0*
1 0 0	*0 0 1*
1 0 1	*0 0 1*
1 1 0	*1 0 0*
1 1 1	*0 0 0*

b) The cycle contains ___*3*____ states.

TECHNISCHE UNIVERSITEIT EINDHOVEN

TENTAMEN DIGITALE SYSTEMEN II
5A010

TOTAL: 8 sheets
of which 4 answersheets

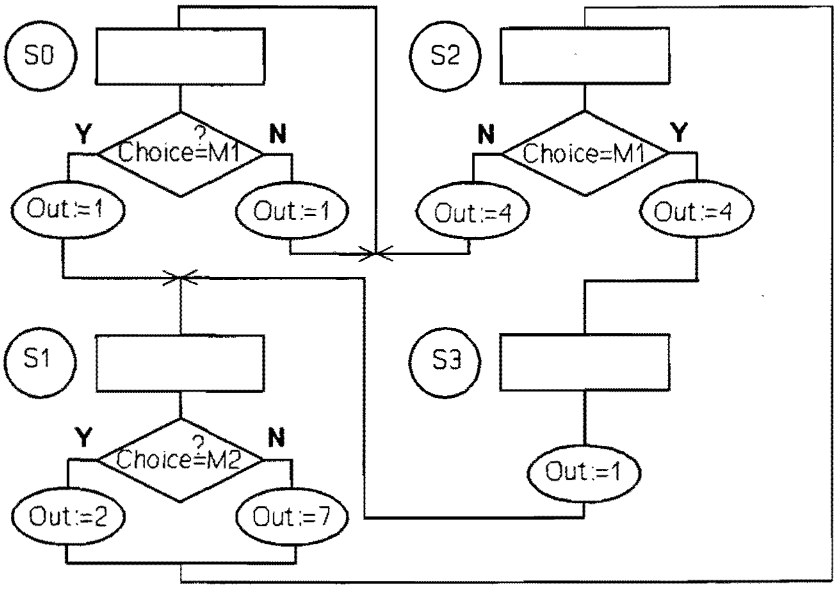
Saturday 10 november 1990
time: 09.00 - 12.00 hours

ANSWERINGSHEET 4

ANSWERSHEETS
NAME:______________________________

I.D. NUMBER:________________________

6. a) Finish the next ASM-chart.



b) In state *S0* : *Out* = ***0,1,4***

In state *S1* : *Out* = ***1***

In state *S2* : *Out* = ***2,7***

In state *S3* : *Out* = ***4***

c) The state table:

	Choice	
State	M1	M2
S 0	***S1,1***	***S0,1***
S 1	***S2,7***	***S2,2***
S 2	***S3,4***	***S0,4***
S 3	***S1,1***	***S1,1***

EXAMINATION "DESIGN OF DIGITAL SYSTEMS"

Wednesday, 1991-03-13

REVISED SOLUTIONS

ex DDS-91

a.
$$\begin{array}{l}
ab + a'c + (bc) = \\
ab + a'c + (bc)\cdot(a + a') = \\
ab + a'c + (abc + a'bc) = \\
(ab + abc) + (a'c + a'bc) = \\
ab + a'c
\end{array}$$

b.
$$\begin{array}{l}
(a \oplus b)' = \\
(ab' + a'b)' = \\
(a' + b)(a + b') = \\
a'a + ab + a'b' + bb' = \\
ab + a'b' = \\
a(b')' + a'(b') = \\
a \oplus b'
\end{array}$$

c.
$$\begin{array}{l}
(a'(b + c'))'\cdot (a + b' + c) \cdot (a'b'c')' = \\
(a'' + b'c) \cdot (a + b' + c) \cdot (a + b + c) = \\
(a + b'c) \cdot ((a + c) + b') \cdot ((a + c) + + b) = \\
(a + b'c) \cdot (a + c + b'b) = \\
(a + b'c) \cdot (a + c) = \\
(a + ab'c + ac + b'c) = \\
a + b'c
\end{array}$$

d.
$$\begin{array}{l}
a + b'c = \\
(a + b') \cdot (a + c) = \\
(a + b' + c)\cdot(a + b' + c')\cdot(a + b + c)\cdot(a + b' + c) \\
(a + b' + c)\cdot(a + b' + c')\cdot(a + b + c) = \\
M_2 \cdot M_3 \cdot M_0 = \\
\Pi(0,2,3)
\end{array}$$

e.
$$\begin{array}{l}
\Sigma\,(0,1,2,4,5,6,7) = \\
(\Sigma\,(\text{remaining minterms}))' = \\
(\Sigma\,(3))' = \\
(a'bc)' = \\
a + b' + c'
\end{array}$$

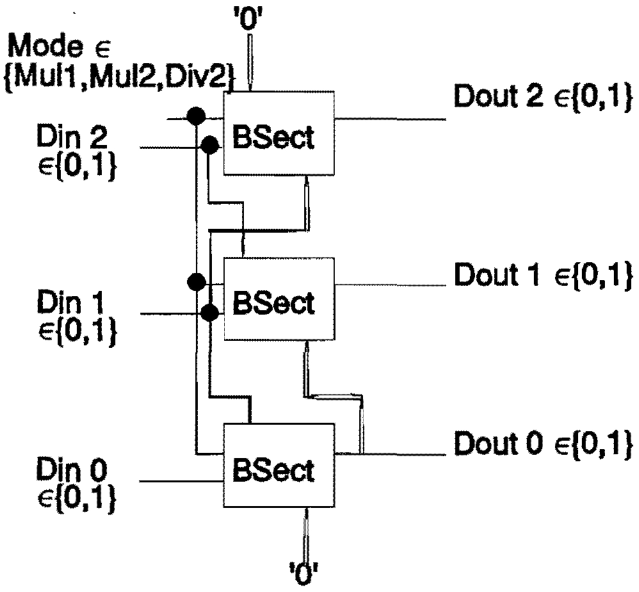
Solution 2

```
2 a. System     MulDiv2;

     TYPE       Modes = (Mul1,Mul2,Div2);
                OctalDigit = 0..7;

     VAR        Mode : Modes;
                Din,Dout:OctalDigit;

     BEGIN
         CASE Mode OF
                Mul1 : Dout := Din;
                Mul2 : Dout := (Din * 2) MOD 8;
                Div2 : Din DIV 2

         END
     END.
```

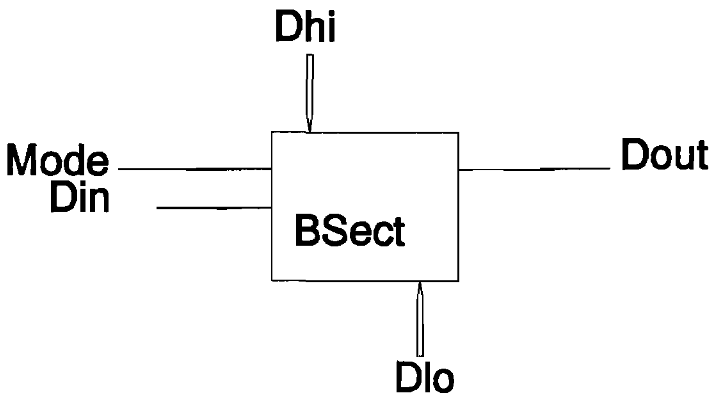
2 b.

	Dout	
Din	Mul2	Div2
1=001	2=010	0=000
2=010	4=100	1=001
3=011	6=110	1=001
5=101	2=010	2=010
	shift left	shift right

2 c.



2 d.



```
TYPE          Modes = (Mul1,Mul2,Div2); Bit = 0..1;
FUNCTION      Bsect (Din,Dhi,Dlo: Bit; Mode: Modes): Bit;
              CASE Mode OF
                      Mul1: BSect: = Din;
                      Mul2: BSect: = Dlo;
                      Div2: BSect: = Dhi;
              END;
```

2 e.

```
System MulDiv2

TYPE        Bit = 0..1;
            Modes = (Mul1,Mul2,Div2);

VAR         Mode : Modes;
            Din2,Din1,Din0 : Bit;
            Dout2,Dout1,Dout0 : Bit;

FUNCTION  BSect (Din,Dhi,Dlo : Bit; Mode : Modes) : Bit;
     CASE.....END; {see 2d}

BEGIN
     Dout2 := BSect(Din2,0,Din1,Mode);
     Dout1 := BSect(Din1,Din2,Din0,Mode);
     Dout0 := BSect(Din0,Din1,0,Mode)
END.
```

Solution 3

3a+b.

State	Next State	Exitation functions
s_2 s_1 s_0	ns_2 ns_1 ns_0	t_2 t_1 t_0
1 0 0	1 1 0	0 1 0
1 1 0	0 1 0	1 0 0
0 1 0	0 1 1	0 0 1
0 1 1	0 0 1	0 1 0
0 0 1	0 0 0	0 0 1
0 0 0	1 0 0	1 0 0
1 0 1	1 0 1	0 0 0
1 1 1	1 1 1	0 0 0

3 c. $t_2 = s_2's_1's_0' + s_2s_1s_0' = \Sigma(0,6)$
$t_1 = s_2's_1s_0 + s_2s_1's_0' = \Sigma(3,4)$
$t_0 = s_2's_1's_0 + s_2's_1s_0' = \Sigma(1,2)$

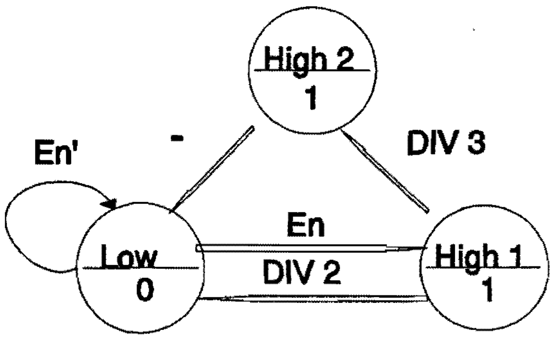
3 d. PLA: see answer sheet.

3 e. MOM applies the sharing of product terms.
As all productterms are minterms, and none of them can be shared (see 3c), no MOM is possible.

3 f. A PAL is cheaper than a PLA because of its fairly fixed structure. A PLA can be advantageous for large, complex circuits with possibilities for product term sharing. In this case we deal with a simple circuit without product term sharing; hence a PAL is to be preferred.

solution 4

4 a. State diagram



4 b. Output patterns

	Pattern	Period length
En'	:000...	1
En.DIV2	:0101...	2
En.DIV3	:011011...	3

4 c.

En : Enable Input - The circuit gives a constant 0 output if not enabled. The 1 behind the name indicates "command input no. 1".

Div2 : The output waveform has a frequency of half the clock frequency. The 1 in front of the name indicates dependency of En : if EN',DIV2 is disabled.

Div3 : The output waveform has a frequency of one third of the clock frequency (see further DIV2). DIV3 = DIV2' is actually a single signal with a double function.

C : Clock, positive edge triggered; disabled for En'.

4 d. We can reduce the register size by combining state and output bits. This is possible with the following state code assignment:

State	code	Current output: Out
Low	00	0
High1	01	1
High2	11	1

We see that Out = s_0

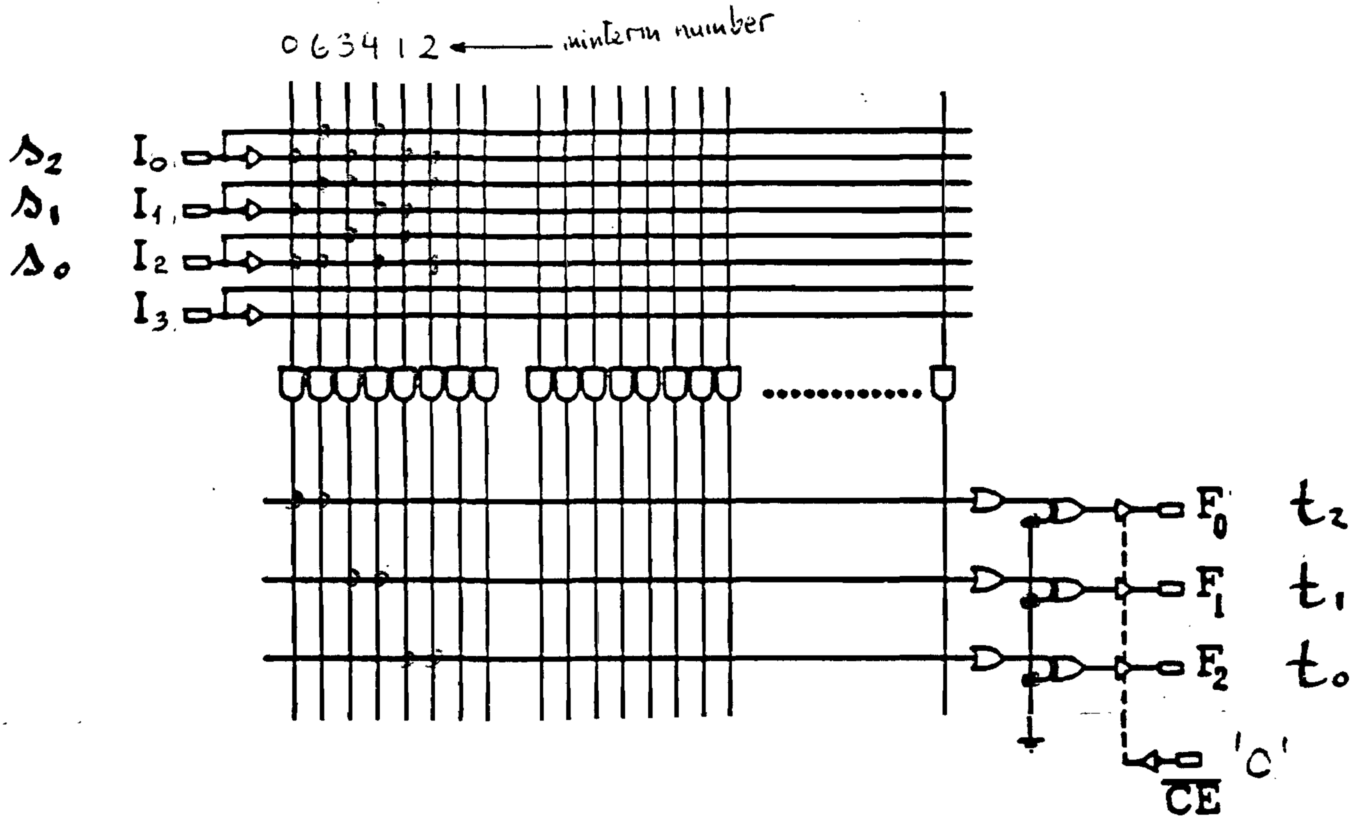
4 e. State table

En	DIV2	s_1s_0	ns_1ns_0	
0	X	00 = Low	00 = Low	
1	X	00 = Low	01 = High1	
X	1	01 = High1	00 = Low	
X	0	01 = High1	11 = High2	
X	X	11 = High2	00 = Low	
X	X	10 = Undef	00 = Low	—>NB!

NB! During power-up and when transient errors occur the system could enter the undefined state "10". By making an unconditional transition to "00" (Low), we make sure that we enter a defined state next.

Student name

ANSWER SHEET FOR PROBLEM 3d



Eindhoven International Institute
1993-01-09

EXAMINATION "DESIGN OF DIGITAL SYSTEMS"

Wednesday, 1992-03-09

SOLUTIONS

ex DDS-92

1 Switching functions

a. $F(w,x,y,z) = w'y'z + wyz + x'yz' + wx'z$

$= \sum (0x01,1x11,x010,10x1)$

$= \sum (0001,0101,1011,1111,0010,1010,1001,1011)$

$= \sum (1,5,11,15,2,10,9,11) = \sum (1,2,5,9,10,11,15)$

b. $G(w,x,y,z) = w(y'z' + xy') + x(z' + y\oplus w) + w'yz + x'y'z'$

$= \sum (wy'z' + wxy' + xz' + xy'w + xyw' + w'yz + x'y'z')$

$= \sum (1x00,110x,x1x0,110x,011x,0x11,x000)$

$= \sum (1000,1100,1100,1101,0100,0110,1100,1110,1100,1101,0110,0111,0011,0111,0000,1000)$

$= \sum (1000,1100,1101,0100,0110,1110,0111,0011,0000)$

$= \sum (8,12,13,4,6,14,7,3,0) = \sum (0,3,4,6,7,8,12,13,14)$

c. After having determined G it is easy to determine G, because the following holds:
$G' = \sum$ (remaining minterms) $= \sum (1,2,5,9,10,11,15)$
This is precisely the sum of minterms for F.
Accordingly:

$$F = G'$$

d. Because $F = G'$, it follows that $F\cdot G = G'\cdot G = 0$.
A fuction table then becomes quite simple:

wxyz	F·G
xxxx	0

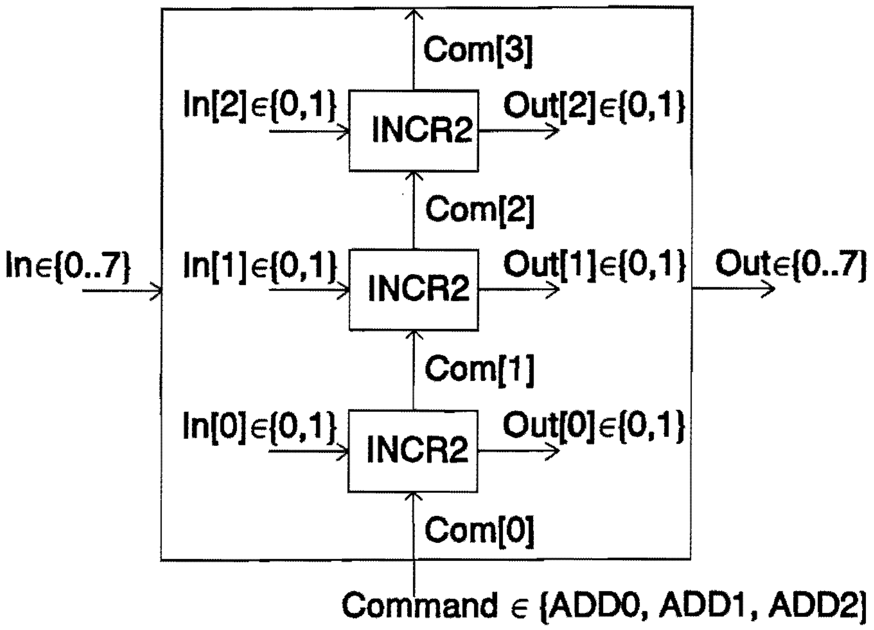
a.

```
TYPE Func       = (ADD0,ADD1,ADD2);
VAR  In, Out    :  0..7;
     Command :  Func;
BEGIN
     CASE Command OF
        ADD0:  Out := In;
        ADD1:  Out := (In + 1) MOD 8;
        ADD2:  Out := (In + 2) MOD 8
     END { CASE }
END.
```

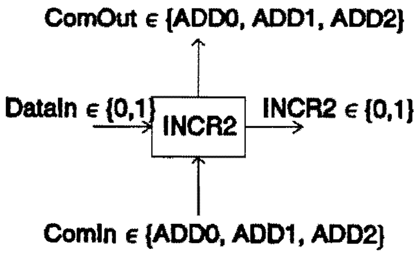
b.

In	Command	Out
0	ADD0	0
0	ADD1	1
0	ADD2	2
1	ADD0	1
1	ADD1	2
1	ADD2	3
2	ADD0	2
2	ADD1	3
2	ADD2	4
-	- - - -	- - -
5	ADD0	5
5	ADD1	6
5	ADD2	7
6	ADD0	6
6	ADD1	7
6	ADD2	0
7	ADD0	7
7	ADD1	0
7	ADD2	1

c.



d.



DataIn	ComIn	INCR2	ComOut
0	ADD0	0	ADD0
0	ADD1	1	ADD0
0	ADD2	0	ADD1
1	ADD0	1	ADD0
1	ADD1	0	ADD1
1	ADD2	1	ADD1

e.

```
TYPE Bit        = 0..1;
     Func       = (ADD0,ADD1,ADD2);

FUNCTION INCR2 (DataIn: Bit; ComIn: Func; VAR ComOut: Func): Bit;
{ Function value = data output }

     CASE ComIn OF

       ADD0:  BEGIN
                   INCR2 := DataIn
                   ComOut := ADD0
                 END;

       ADD1:  BEGIN
                   INCR2 := (DataIn + 1) MOD 2;
                   IF DataIn = 0
                     THEN ComOut := ADD0
                     ELSE ComOut := ADD1        { Carry! }
                 END;

       ADD2:  BEGIN
                   INCR2 := DataIn;
                   ComOut := ADD1
                 END

     END { CASE and function body };
```

f.

```
TYPE Bit        = 0..1;
     Func       = (ADD0,ADD1,ADD2);

FUNCTION INCR2 ( . . . . ): Bit;
     -- see solution for problem e. --
VAR  In, Out    :  ARRAY [0..2] OF Bit;
     Com        :  ARRAY [0..3] OF Func;
     i          :  0..2;
BEGIN
     FOR i := 0 TO 2 DO
       Out [i] := INCR2 (In[i]; Com[i]; Com[i+1])
END.
```

3 Programmed Logic Array

a. We derive from the PLA:

$$f_1 = w'x'y + w'x + wxy' + wx'z = \sum(001x, 01xx, 110x, 10x1)$$
$$f_2 = w'x'y' + w'x + wxy + wx'z = \sum(000x, 01xx, 111x, 10x1)$$

Accordingly:

$$f_1 = \sum(2,3,4,5,6,7,12,13,9,11) = \sum(2,3,4,5,6,7,9,11,12,13)$$
$$f_2 = \sum(0,1,4,5,6,7,14,15,9,11) = \sum(0,1,4,5,6,7,9,11,14,15)$$

b. The product of maxterms can be found from the missing minterms:

$$f_1 = \Pi(0,1,8,10,14,15)$$
$$f_2 = \Pi(2,3,8,10,12,13)$$

c. The dual function f_{1d} can be found by exchanging the + operators and (not explicitly shown) • operators:

$$\begin{aligned} f_{1d} &= (w' + x' + y)\cdot(w' + x)\cdot(w + x + y')\cdot(w + x' + z) \\ &= \Pi(110x, 10xx, 001x, 01x0) \\ &= \Pi(12,13,8,9,10,11,2,3,4,6) = \Pi(2,3,4,6,8,9,10,11,12,13) \end{aligned}$$

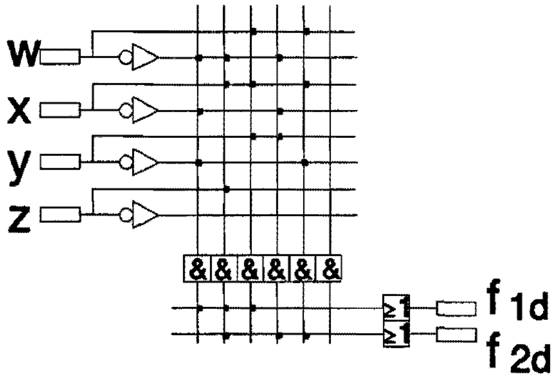
Accordingly:

$$f_{1d} = \sum(0,1,5,7,14,15)$$

In a similar way we find:

$$f_{2d} = \sum(2,3,5,7,12,13)$$

d. The PLA for f_{1d} and f_{2d} is shown below:



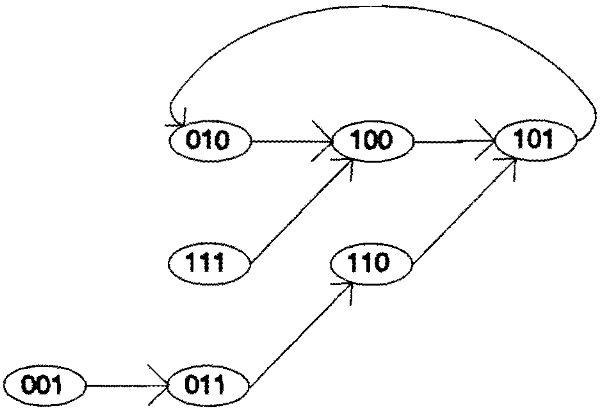
a. According to the given block diagram the following functions hold for the inputs of the D flip-flops. Because we have applied D flip-flops, these are also the next state functions:

$D_2 = ns_2 = s_1 + s_0'$
$D_1 = ns_1 = s_0 \, . \, (s_2' + s_1')$
$D_0 = ns_0 = s_2s_0' + s_1's_2'$

b. By means of these functions we can simply fill in the state table:

Current state	Exitation function	Next state
$s_2s_1s_0$	$D_2D_1D_0$	$s_2s_1s_0$
0 0 0	1 0 1	1 0 1
0 0 1	0 1 1	0 1 1
0 1 0	1 0 0	1 0 0
0 1 1	1 1 0	1 1 0
1 0 0	1 0 1	1 0 1
1 0 1	0 1 0	0 1 0
1 1 0	1 0 1	1 0 1
1 1 1	1 0 0	1 0 0

c. Below the complete state diagram is drawn.
We can clearly see that the cycle consists of 3 states: 001, 010 and 100.



Appendix B

Solutions to Problems

2.1. a.

$$n = \lceil \log_2 99 \rceil = \lceil 6.62\ldots \rceil = 7$$

b. Unused code words:

$$2^n - \#S = 128 - 100 = 28$$

c. # Coding schemes:

$$\frac{(2^n n)!}{(2^n - \#S)!} = \frac{128!}{28!}$$

$$= 29 * 30 * 31 * \ldots. * 128 \approx 10^{186}$$

d. Max # elements in set with 7-bit coding:

$$2^7 = 128$$

e. # Coding schemes with 128 elements:

$$\frac{(2^n)!}{(2^n - \#S)!} = \frac{128!}{0!} = 128! \approx 4 * 10^{215}$$

2.2. a.

$$n = \log_2 4 = 2$$

b. # Coding schemes:

$$\frac{(2^n)!}{(2^n - \#S)!} = (2^n)! = 4! = 24$$

c. All coding schemes:

{00,01,10,11}, {00,01,11,10}, {00,10,01,11}, {00,10,11,01},
{00,11,01,10}, {00,11,10,01}
{01,00,10,11}, {01,00,11,10}, {01,10,00,11}, {01,10,11,00}
{01,11,00,10}, {01,11,10,00}
{10,00,01,11}, {10,00,11,01}, {10,01,00,11}, {10,01,11,00},
{10,11,00,01}, {10,11,01,00}
{11,00,01,10}, {11,00,10,01}, {11,01,00,10}, {11,01,10,00},
{11,10,00,01}, {11,10,01,00}

All schemes are different. If no weights are assigned to the bits, (01 is equal to 10) 3 bits are needed; code words = 000, 001, 011, 111.
Here also: 4 codewords assigned to 4 elements:
# Coding schemes:

$$4! = 24$$

2.3. a. Greatest value if all bits are 1:

$$2^{n-1} + 2^{n-2} + \ldots. + 2^2 + 2^1 + 2^0$$
$$= 1 * (1 - 2^n) / (1 - 2) = 2^n - 1$$

(formula of geometrical series)

b.

$n = 8$: $\quad$ *greatest value* $= 2^8 - 1 = 255_{10}$

c.

$n = 16$: $\quad$ *greatest value* $= 2^{16} - 1 = 65535_{10}$

2.4.

a. 10110101 = ((((((1 * 2+0) * 2+1) * 2+0) * 2+1) * 2+0) * 2+1 =

b. 01101100 = (((((1 * 2+1) * 2+0) * 2+1) * 2+1) * 2+0) * 2+0 =

2.5.

a.

216/4 = 54 rem. 0
54/4 = 13 rem. 2
13/4 = 3 rem. 1
3/4 = 0 rem. 3

or:

div 4	mod 4
216	0
54	2
13	1
3	3

$216_{10} = 3120_4$

b.

div 4	mod 4
99	3
24	0
6	2
1	1

$99_{10} = 1203_4$

c./d. We split each hex digit in 2 quaternary digits:

base 16	base 2	base 4
0	00 00	0 0
1	00 01	0 1
2	00 10	0 2
3	00 11	0 3
4	01 00	1 0
5	01 01	1 1
6	01 10	1 2
7	01 11	1 3
8	10 00	2 0
9	10 01	2 1
A	10 10	2 2
B	10 11	2 3
C	11 00	3 0
D	11 01	3 1
E	11 10	3 2
F	11 11	3 3

Hence:

c. $A7_{16} = 2213_4$
d. $83_{16} = 2003_4$

2.6.

a.

div 2	mod 2
10985	1
5492	0
2746	0
1373	1
686	0
343	1
171	1
85	1
42	0
21	1
10	0
5	1
2	0
1	1

$10985_{10} = 10101011101001_2$

b.

div 5	mod 5
278	3
55	0
11	1
2	2

$278_{10} = 2103_5$

2.7.

```
1 1 1 1 1
  0 1 1 0 1 1
  1 1 0 1 1 0 +
---------------
1 0 1 0 0 0 1
```

The result is one bit longer than the operands.

3.1.

```
VAR A,B:   0..15;
   MUL:   0..255;
BEGIN
   MUL: = A*B
END
```

3.2.

```
VAR A,B:  -9..9;
   SEL:   (Add, Subtract);
   RES:  -19..19;
BEGIN
    IF   (SEL = Add)
        THEN  RES : = A + B
        ELSE  RES : = A - B
END
```

3.3.

```
TYPE    Inval    =  -9..9;
        Outval   =  -19..19;
        Choice   =  (Add, Subtract);
VAR A,B:  Inval;
   SEL:  Choice;
   RES:  Outval;
FUNCTION ADD (X,Y: Inval) : Outval;
   BEGIN ADD : = X + Y  END;
FUNCTION SUB (X,Y: Inval) : Outval;
   BEGIN SUB : = X - Y  END;
FUNCTION MUX (In0,In1 : Outval; S : Choice) : Outval;
   BEGIN
     IF   (S = Add)
       THEN MUX  : = In0
       ELSE MUX  : = In1
     END;
BEGIN
   RES : = MUX (ADD (A,B), SUB (A,B), SEL)
END.
```

.4. CMP4 module:

```
TYPE Quit       =  0..3;
      CmpRes  =  (greater, equal, less)
FUNCTION
      CMP4 ( u,v: Quit; Prev: CmpRes) : CmpRes;
BEGIN
  IF   (Prev = equal)
   THEN IF (u > v)
     THEN CMP4 : = greater
     ELSE IF (u = v)
       THEN CMP  : = equal
       ELSE CMP  : = less
   ELSE CMP  : = Prev;
END;
```

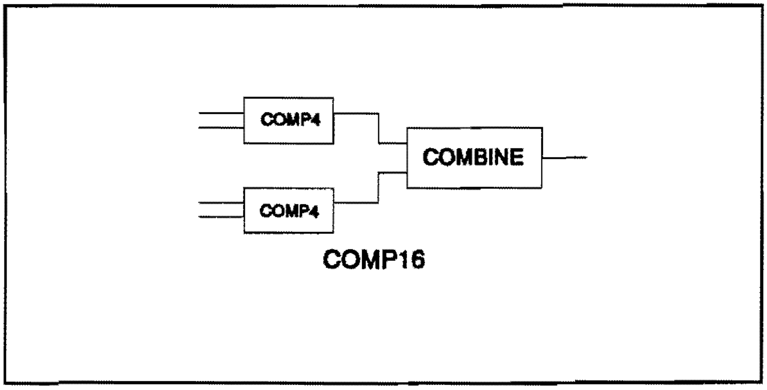
Whole system:

```
VAR A,B:  ARRAY [0..3] OF Quit;
   OUT, Tmp:  CmpRes;
   I  :  Integer;
BEGIN
     Tmp : = equal;
     FOR I : = 3 DOWNTO 0 DO
        Tmp : = CMP4 ( A[I], B[I], Tmp);
     OUT   : = Tmp
END.
```

.5.

ci u v	u + v + ci	SM= (u+v+ci)MOD2	CO= (u+v+ci)DIV2
0 0 0	0	0	0
0 0 1	1	1	0
0 1 0	1	1	0
0 1 1	2	0	1
1 0 0	1	1	0
1 0 1	2	0	1
1 1 0	2	0	1
1 1 1	3	1	1

3.6. Both inputs have as domain {0..15}
One-to-one mapping to $\{0..3\}^2$ is possible.
A 2-layer tree structure could therefore be possible:



Description of COMP4:

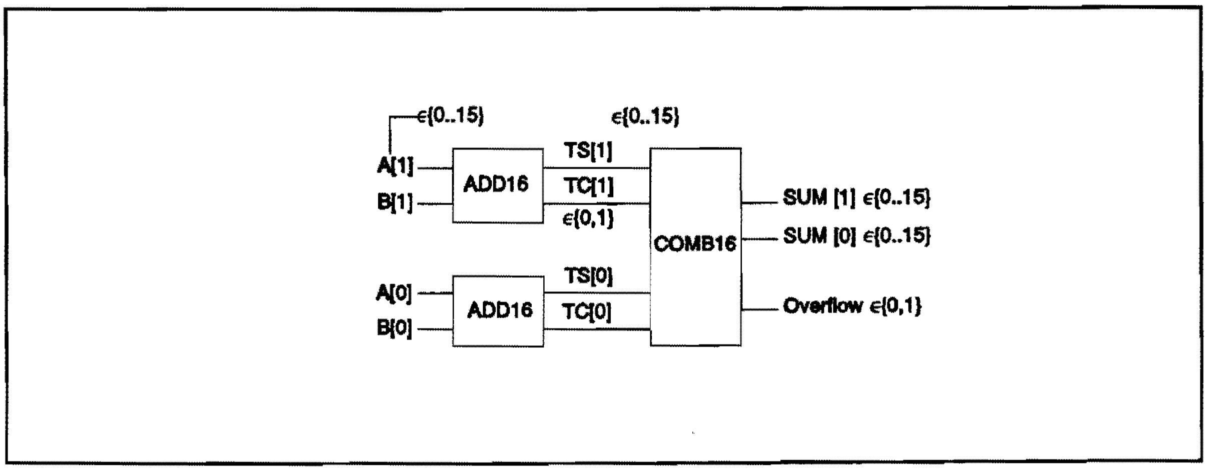
```
TYPE Hit = 0..15;
       Quit = 0..3;
       CompRes = (greater, equal, less);
FUNCTION
COMP4( u,v: Quit): CompRes;
BEGIN
  IF (u>v)
  THEN COMP4 := greater
  ELSE IF (u=v)
   THEN COMP4 := equal
   ELSE COMP4 := less
END;
```

Description of the new COMP16:

```
TYPE CodedHit = ARRAY[0..1] OF Quit;
FUNCTION
  COMP16( u,v: CodedHit): CompRes;
  BEGIN
    COMP16 : = COMBINE(COMP4(u[1],v[1]), COMP4(u[0],v[0]))
  END;
```

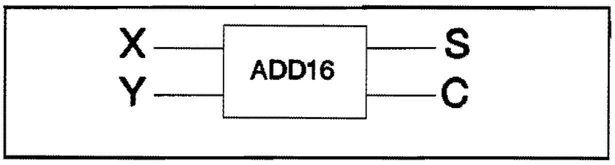
Each COMP16 in Fig. 3.56 is replaced by this 2-layer tree structure. Resulting structure: corresponds to Fig. 3.44 (three layers).

.7. <u>2-layer tree structure</u> (we expand to 3 later):
ADD16 : adds 2 numbers 0..15
COMB16: combines result of these



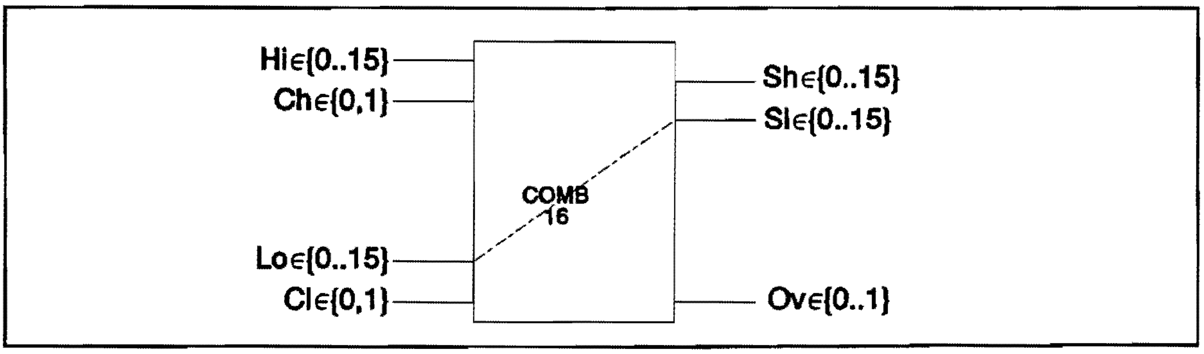
TS[i] = Temporary Sum [i] ∈ {0..15}
TOV[i] = Temporary Carry [i] ∈ {0..15}

<u>ADD16 module:</u>



```
TYPE Hit = 0..15;                         }
       Bit = 0..1;                        }
VAR   A,B,TS,SUM,TOV: ARRAY[0,1] OF Hit;  }global variables
       TC: ARRAY[0,1] OF Bit;             }
       OverFlow: Bit;                     }
PROCEDURE
  ADD16( X,Y: Hit; VAR S: Hit; VAR C: Bit);
  BEGIN
    S := (X+Y)MOD16; (* sum *)
    C := (X+Y)DIV16    (* overflow *)
END;
```

COMB16 module:



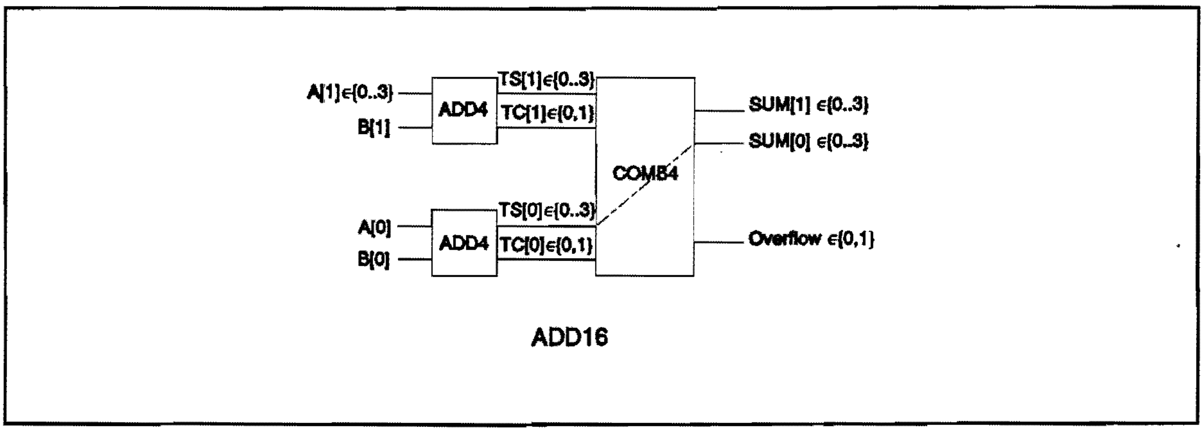
```
PROCEDURE
 COMB16( Hi,Lo: Hit; Ch,Cl: Bit;
  VAR Sh,Sl: Hit; VAR Ov: Bit);
 BEGIN
  Sl := Lo;  (* Least significant nibble *)        } nibble = 4-bit number
  Sh := (Hi+Cl)MOD16;
  Ov := Ch+(Hi+Cl)DIV16;
END;
```

N.B.! Sl: = Lo passes Lo unmodified through COMB16.

Combination of modules in 2 layers:

```
BEGIN
 ADD16(A[1],B[1],TS[1],TC[1]);
 ADD16(A[0],B[0],TS[0],TC[0]);
 COMB16(TS[1],TS[0],TC[1],TC[0],SUM[1],SUM[0],OverFlow)
END.
```

a/c Development of 3 layers: ADD16 is split in 2 layers

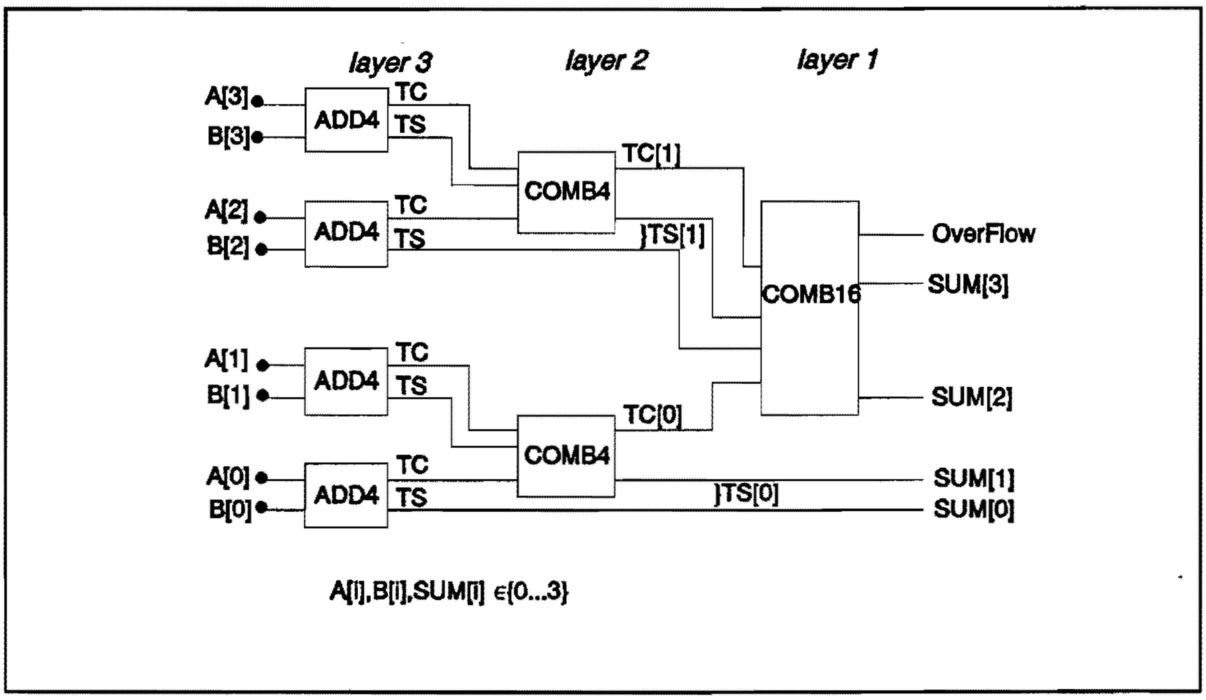


(ADD16 has been built up in a similar way as the 2-layer adder we started with ADD256; however, the domains of the variables have been restricted)

Behavioural description:

```
TYPE Qit = 0..3;
      Bit = 0..1;
VAR   A,B,TS,SUM,TOV: ARRAY[0,1] OF Qit;
      TC: ARRAY[0,1] OF Bit;
PROCEDURE
      ADD4( X,Y: Qit; VAR S: Qit; VAR C: Bit);
BEGIN
      S := (X+Y)MOD4;  (* sum *)
      C := (X+Y)DIV4  (* overflow *)
END;
PROCEDURE
 COMB4( Hi,Lo: Qit; Ch,Cl: Bit;
  VAR Sh,Sl: Qit; VAR Ov: Bit);
BEGIN
  Sl := Lo;  (* Least significant part *)
  Sh := (Hi+Cl)MOD4;
  Ov := Ch+(Hi+Cl)DIV4
END;

BEGIN
 ADD4(A[1],B[1],TS[1],TC[1]);
 ADD4(A[0],B[0],TS[0],TC[0]);
 COMB4(TS[1],TS[0],TC[1],TC[0],SUM[1],SUM[0],OverFlow)
END.
```

Block diagram of whole system:



Remarks: - Signals which are passed through COMB4 or COMB16 without modification have been drawn outside these blocks. For this reason the resulting tree-structure is not quite pure.

.7.

b. → - Blocks on layer 1 and layer 2 have the same functionality, but are not identical. This is in contradiction with the architecture scheme covered earlier (Fig. 3.34 - all F blocks are identical)

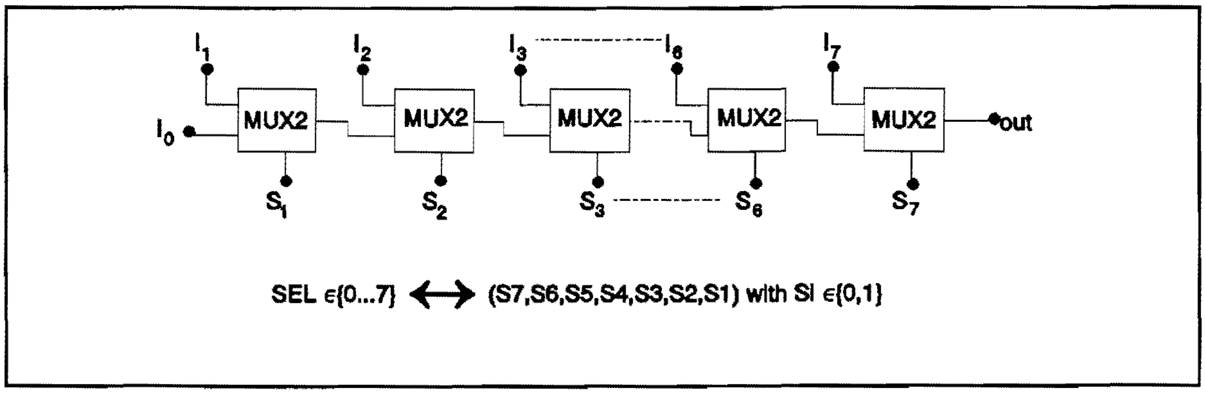
3.8. Block diagram: given in Fig. 3.41

Behavioural description:

```
TYPE Bit=(0,1);
VAR  SEL: ARRAY[0..2] OF Bit;
     I0,I1,I2,I3,I4,I5,I6,I7,OUT: AnyType;

FUNCTION MUX2( ....): AnyType:
 (* Behavioural description: see Fig. 3.38 *)

BEGIN (* 8-input multiplexer *)
 OUT:=  MUX2( MUX2( MUX2(I7,I6,SEL[0]),
                    MUX2(I5,I4,SEL[0]),
                    SEL[1]),
              MUX2( MUX2(I3,I2,SEL[0]),
                    MUX2(I1,I0,SEL[0]),
                    SEL[1]),
              SEL[2]);
END
```

3.9. Yes, it is possible to realize an 8-input multiplexer by means of an iterative architecture; its derivation, however, is slightly different from the one covered so far.

Block diagram:



SEL has been coded as a "1-out-of-7 code":

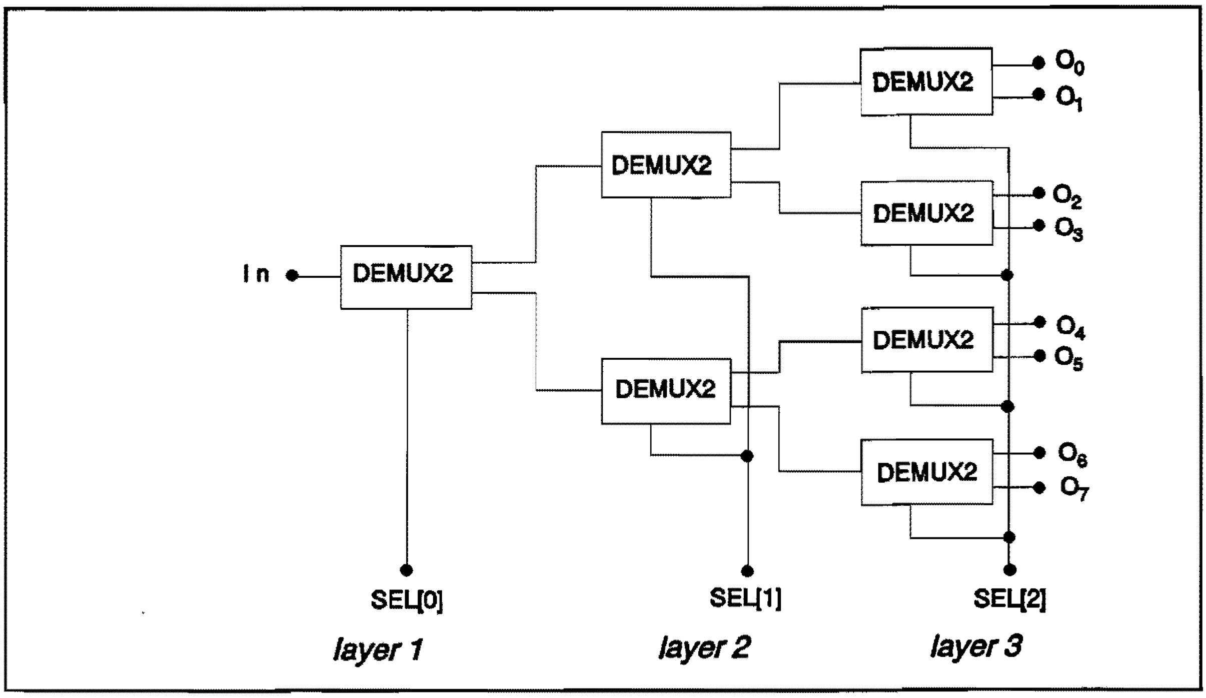
SEL	S_7	S_6	S_5	S_4	S_3	S_2	S_1
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	0	0	1	0
3	0	0	0	0	1	0	0
4	0	0	0	1	0	0	0
5	0	0	1	0	0	0	0
6	0	1	0	0	0	0	0
7	1	0	0	0	0	0	0

We see that the "0"-bits to the right of the "1" in each code-word are of no importance; they could have been replaced by any other bit pattern.

.10.

```
VAR In,O0,O1,O2,O3,O4,O5,O6,O7: AnyType;
    SEL: 0..7;
BEGIN
  CASE SEL OF
    0 : O0 := In;
    1 : O1 := In;
    2 : O2 := In;
    3 : O3 := In;
    4 : O4 := In;
    5 : O5 := In;
    6 : O6 := In;
    7 : O7 := In
  END (* CASE *)
END
```

.11. Realization of an 8-output demultiplexer in a 3-layer tree structure:



The tree structure is the opposite of the one for the multiplexer: its "root" on layer 1 is at the input, and its "branches" (with the highest number of circuits) at the output.

.12. Behavioural description

```
TYPE CompRes = (greater,equal,less);
FUNCTION
  COMB(Cmp3,Cmp2,Cmp1,Cmp0: CompRes): CompRes;
  BEGIN
    IF (Cmp3 = equal)
      THEN
        IF (Cmp2 = equal)
          THEN
            IF (Cmp1 = equal)
              THEN COMB := Cmp0
              ELSE COMB := Cmp1
          ELSE COMB := Cmp2
      ELSE COMB := Cmp3
END;
```

Cmp3..Cmp0 are the outputs of the COMP4 circuits in Fig. 3.47 from top to bottom, each having a value ∈ {greater, equal, less}.

4.1.

a: a'b + a·(b'+c) + a·b'c = {P3} a'b + a·(b'+c+b'c) = {Th4} a'b + a·(b'+c) *qed.*

b: abc' + a'bc + a'bc' + abc + a'b'c + ab'c = {Th2} (abc'+abc) + (a'bc+a'bc') + (abc+ab'c)
+ (a'bc+a'b'c) = {P3,P5} ab·1 + a'b·1 + ac·1 + a'c·1 = {P3,P5} b + c *qed.*

c: (((xy)'z)' + x'yz')' = {Th7} (((xy)'z)')'·(x'yz')' = {Th7} ((xy)'z)·(x''+y'+z'') = {Th7,Th6}
((x'+y')z)·(x+y'+z) = (x'+y')·z·(z+(x+y')) = {Th4} (x'+y')·z *qed.*

d: x +x'y + z(x+y) = {Th5} x + y + z(x+y) = {Th4} x+y *qed.*

e: (a+c')·(a+b)·(b+c) = {P4,P5} (a+c')·(a+b+c'·c)·(b+c) = {P3}
(a+c')·(a+b+c')·(a+b+c)·(b+c) = {Th4} (a+c')·(b+c) *qed.*

f: xy + x'z + yz = {P4,P5} xy + x'z + yz(x+x') = xy + xyz + x'z + x'yz = {Th4} xy + x'z *qed.*
We see from this deviation that the dual expression of e is f.

g: wx + xy + x'z' + wy'z = {f: add redundant term wz'} xy + wx + x'z' + wz' + wy'z = {Th5}
xy + wx + x'z' + wz' + wy' = {f: redundant term wz'} xy + wx + x'z' + wy'
= {f:redundant term wx} xy + x'z' + wy' *qed.*

4.2.

m_{10} = v'·w·x'·y·z'
m_{10}' = (v'·w·x'·y·z')' = v''+w'+x''+y'+z'' (De Morgan)
= v+w'+x+y'+z = m_{10} according to standard maxterm numbering)

4.3.

a: Observe that f_2 already exists in its sum-of-minterms form. For this reason we shall also convert f_1 into this form:
f_1 = (xy'+x'y)' = (x'+y)·(x+y') = x'x + x'y' + xy + yy' = 0+x'y'+xy = x'y'+xy = f_2

b: We observe that:

$$w\,\overline{x}\,\overline{z} = w\,\overline{x}\,(y+\overline{y})\overline{z} = w\,\overline{x}\,y\,\overline{z} + w\,\overline{x}\,\overline{y}\,\overline{z} = 1010 + 1000$$
$$= 10x0 \qquad (x\ can\ be\ 0\ or\ 1)$$

Using this technique we can write:
f_1 = ∑(10x0, 11xx, 010x) = ∑(8,10,12,13,14,15,4,5) = ∑(4,5,8,10,12,13,14,15)
f_2 = Π(x0x1, 00xx,0x1x) = Π(1,3,9,11,0,1,2,3,2,3,6,7) = Π(0,1,2,3,6,7,9,11)

We write f_2 as a sum of missing minterms:
f_2 = ∑(4,5,8,10,12,13,14,15) = f_1

c: If we write:
f_1' = ∑(x0x, 0x1) = ∑(0,1,4,5,1,3) = ∑(0,1,3,4,5)
then f_1 is the sum of missing minterms:
f_1= ∑(2,6,7) = ∑(010, 110,111) = x'yz' + xyz' + xyz = f_2

d: We write f_1 as a product of maxterms:
f_1 = Π(01x, 000, 001) = Π(0,1,2,3)
The following now holds:
f_1 = ∑(4,5,6,7) = ∑(100,101,110,111) = ∑(10x, 11x) = ∑(1xx) = x = f_2

4\.

a\. $T = \sum(000x, x1x1, 0x01) = \sum(0,1, 5,7,13,15, 1,5)$

$= \sum(0,1,5,7,13,15)$

b: We derive:

$T' = \sum(2,3,4,6,8,9,10,11,12,14)$, accordingly:

$T = \Pi(2,3,4,6,8,9,10,11,12,14)$

c: The expressions for T' are the sums of missing min- or maxterms:

$T' = \sum(2,3,4,6,8,9,10,11,12,14)$

$= \Pi(0,1,5,7,13,15)$

d: A <u>dual</u> function is obtained by replacing:

$+$ by $\cdot$

$\cdot$ by $+$

1 by 0

0 by 1

In the given function:

$T_d = (w'+x'+y')\cdot(x+z)\cdot(w'+y'+z)$

We can now directly derive the product-of-maxterms:

$T_d = \Pi(111x, x0x0, 1x10) = \Pi(14,15,0,2,8,10,10,14)$

$= \Pi(0,2,8,10,14,15)$

Accordingly:

$T_d = \sum(1,3,4,5,6,7,9,11,12,13)$

<u>Remark</u>:

The product-of-maxterms of the dual function can also be directly derived from T's minterms. The following holds:

The maxterms of T_d are equal to the 1's complement of the minterms of T. The 1's complement is given by:

$(2^n - 1 - \text{minterm})$

where n is the function's number of variables.

Check that this method gives the correct results in this problem.

An analog expression can be formed for the sum of minterms.

5\.

a\. We start with the product of maxterms, which is easier:

b: $T = \Pi(1,2,3,5,6,10,12)$

a: $T = \sum(0,4,7,8,9,11,13,14,15)$

c: $T' = \sum(1,2,3,4,5,10,12) = \Pi(0,4,7,8,9,11,13,14,15)$

d: $T_d = \sum(0,1,2,4,6,7,8,11,15) = \Pi(3,5,10,11,12,13,14)$

b\.

a: $T = \sum(5,6,7,9,11)$

b: $T = \Pi(0,1,2,3,4,8,10,12,13,14,15)$

c: $T' = \sum(0,1,2,3,4,8,10,12,13,14,15) = \Pi(5,6,7,9,11)$

d: $T_d = \sum(4,6,8,9,10) = \Pi(0,1,2,3,5,7,11,12,13,14,15)$

c\.

a: $T = \sum(2,3,6,7,9,10,11,13,14,15)$

b: $T = \Pi(0,1,4,5,8,12)$

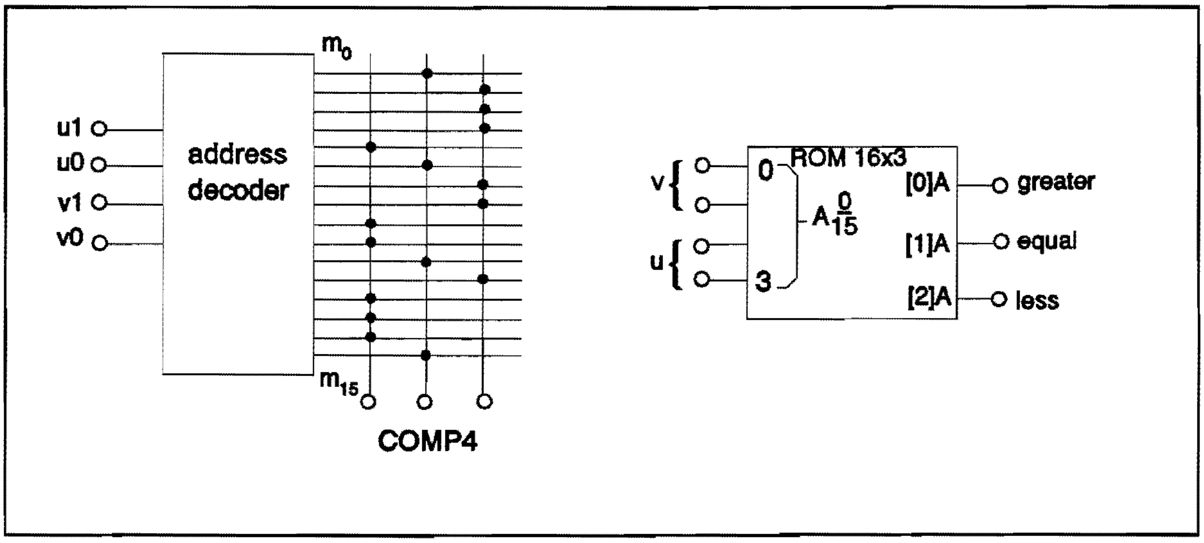
c: $T' = \sum(0,1,4,5,8,12) = \Pi(2,3,6,7,9,10,11,13,14,15)$

d: $T_d = \sum(0,1,2,4,5,6,8,9,12,13) = \Pi(3,7,10,11,14,15)$

5.1. The coding of the comparator has been specified in the problem; the input coding has not, however. We make the apparent choice of using 2-bit binary numbers.
The truth table is as follows:

u_1 u_0 v_1 v_0	COMP4	u_1 u_0 v_1 v_0	COMP4
0 0 0 0	010	1 0 0 0	100
0 0 0 1	001	1 0 0 1	100
0 0 1 0	001	1 0 1 0	010
0 0 1 1	001	1 0 1 1	001
0 1 0 0	100	1 1 0 0	100
0 1 0 1	010	1 1 0 1	100
0 1 1 0	001	1 1 1 0	100
0 1 1 1	001	1 1 1 1	010

This table can be directly translated to a ROM implementation. We agree to use u_1, u_0, v_1 and v_0 as the ROM address bits in decreasing order of significance.
The ROM implementation and the corresponding IEC symbol is shown below:



5.2.

a. We first add variable a:

$bd = bd \cdot 1 = bd(a'+a) = a'bd + abd$

c is afterwards added to each term:

$bd = a'bd \cdot 1 + abd \cdot 1 = a'bd(c'+c) + abd(c'+c) = a'bc'd + a'bcd + abc'd + abcd$

b. We just specify the result:

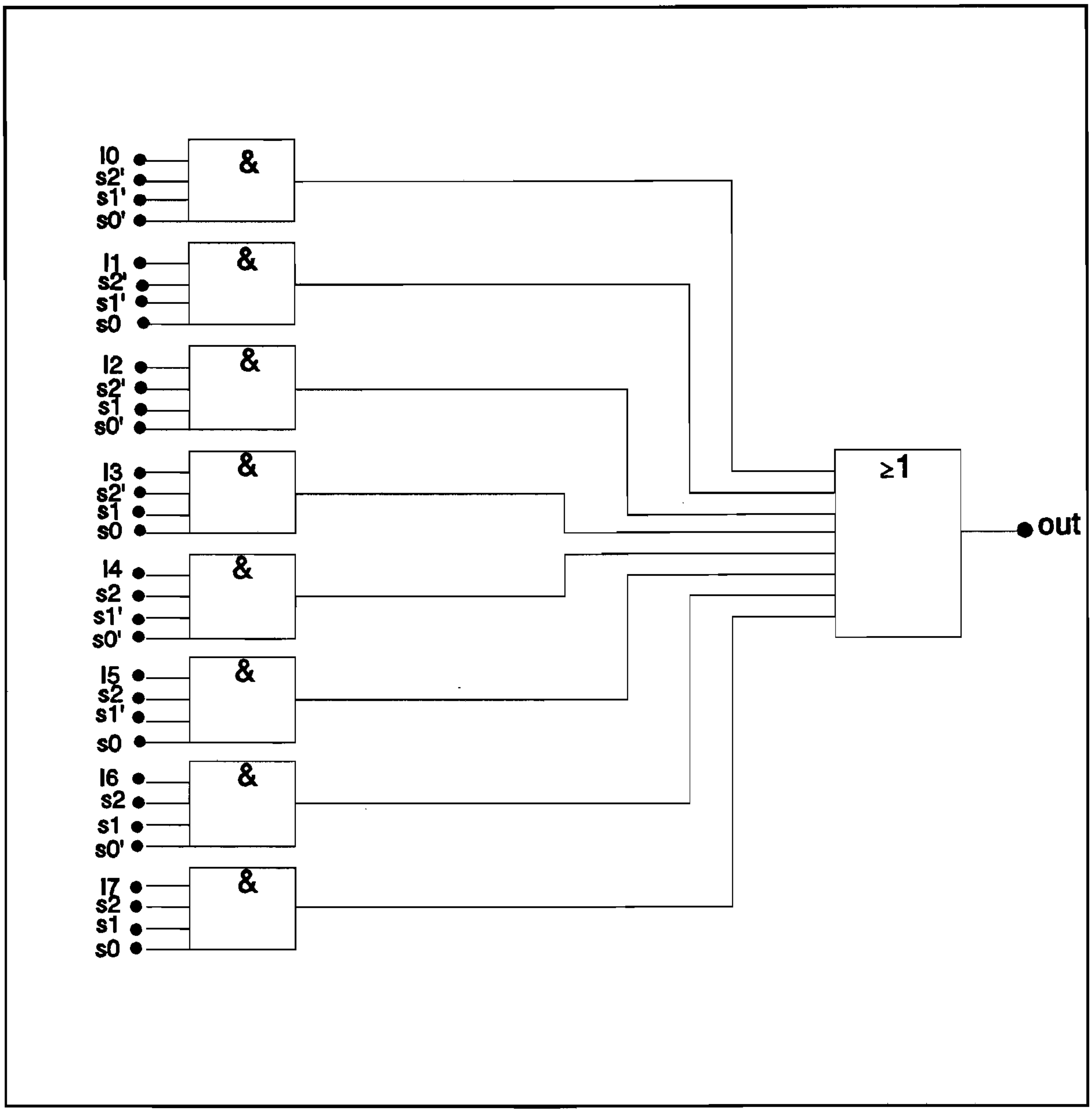
$a = ab'c' + ab'c + abc' + abc$

5.3. The function table comprises 8 product terms only. We code SEL with the binary triple (s_2, s_1, s_0). The multiplexer function can now be described by the following sum-of-product-terms:

$OUT = I_0s_2's_1's_0' + I_1s_2's_1's_0 + I_2s_2's_1s_0' + I_3s_2's_1s_0 + I_4s_2s_1's_0' + I_5s_2s_1's_0 + I_6s_2s_1s_0' + I_7s_2s_1s_0$

This expression can be realized with 8 and-gates and 1 or-gate.

The resulting block diagram is shown below:



We directly derive from the block diagram of Fig. 2.28:

$F = x_2x_1' + x_2'x_1 + x_0$

The function table in Fig. 5.27 specifies the function f as the following sum of product terms:

$f = x_1'x_0 + x_2'x_1 + x_2x_1' + \underline{x_2x_1x_0}$

The don't care term has been underlined.
We show that F = f by adding variables to f; we start with x_0 because the other terms already exist in f: $x_2'x_1 + x_0 = x_2'x_1 + x_0\cdot(x_1'+x_1) = x_2'x_1 + x_1'x_0 + x_1x_0\cdot(x_2'+x_2)$
$= x_2'x_1\cdot(1+x_0) + x_1'x_0 + x_2x_1x_0 = x_2'x_1 + x_1'x_0 + x_2x_1x_0$
so that $F = x_2x_1' + x_2'x_1 + x_1'x_0 + x_2x_1x_0$

We see that F = f if the don't care term gives a function value f(1,1,1) = 1. Accordingly the don't care term $x_2x_1x_0$ is included in the diagram.

Remark:
We can also compare the functions by rewriting them in a sum-of-minterms form, or by converting the function table to a truth table.

5.5. If x_2 is used as selection variable, we can rewrite the truth table as follows:

x_2	x_1 x_0	f	
0	0 0	0	
	0 1	1	=> f_0
	1 0	1	
	1 1	1	

x_2	x_1 x_0	f	
1	0 0	0	
	0 1	1	=> f_1
	1 0	0	
	1 1	-	

We derive:

$f_0 = x_1 + x_0$
$f_1 = x_1' + \underline{x_1 x_0}$

If x_0 is used as selection variable, we get:

x_0	x_2 x_1	f	
0	0 0	0	
	0 1	1	=> f_0
	1 0	1	
	1 1	0	

x_0	x_2 x_1	f	
1	0 0	1	
	0 1	1	=> f_1
	1 0	1	
	1 1	-	

We derive:

$f_0 = x_1 \oplus x_1$
$f_1 = x_2' + x_1' + \underline{x_2 x_1}$

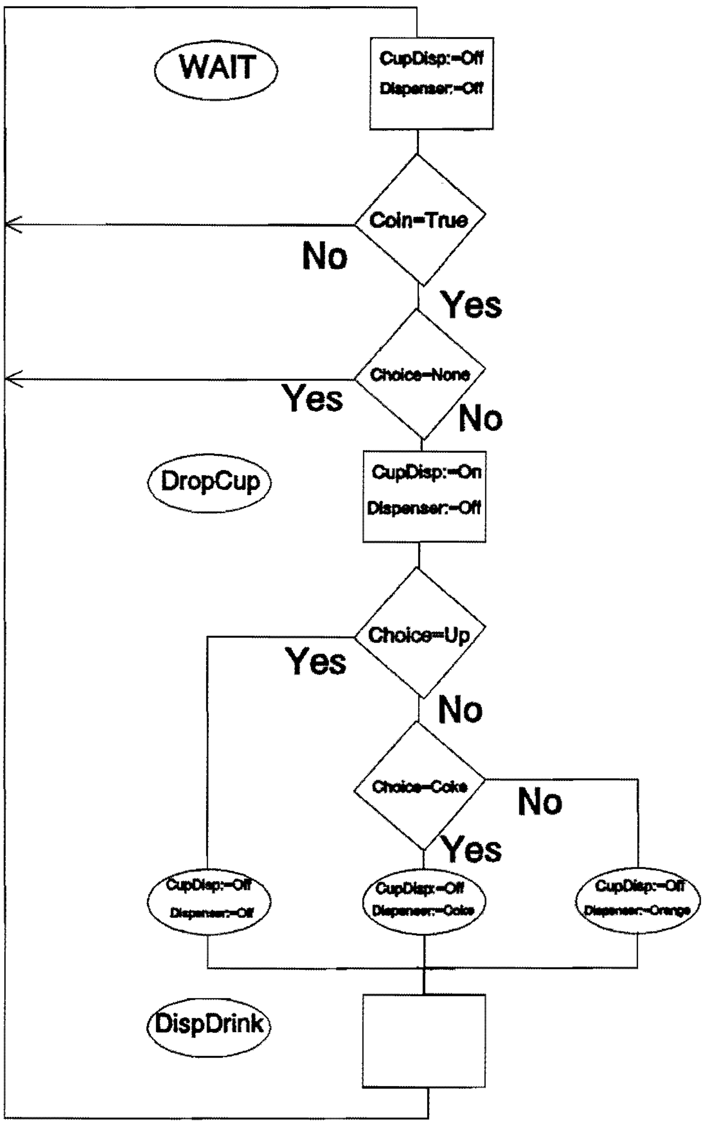
6.1. The missing parts are shown below:

```
CASE State OF
  :
  :
  2D: IF Number = 4
    THEN  State:=3D
    ELSE  State:=0D;
  3D: IF Number = 8
    THEN  State:=4D
    ELSE  State:=0D;
  :
END;
```

2. The Pascal behavioural description is given below:

```
SYSTEM Mod_5_Counter;
TYPE RANGE = 0..4;
        MODE = (start, stop);
VAR   Clockpuls: EVENT;
        Countermode: MODE;
        CounterOutput: RANGE;
BEGIN
  REPEAT
    WAIT_FOR_EVENT(ClockPulse);
    IF Countermode = start
      THEN CounterOutput:=(CounterOutput+1) MOD 5;
  FOREVER
END.
```

6.3. The simplified ASM chart is shown below. The output boxes for the states "Wait" and "DispDrink" have been removed; instead, the state boxes for "Wait" and "DropCup" have got output value specifications.



.4. Here we shall only give a state table with input conditions and not with input values. The following state abbreviations have been introduced:

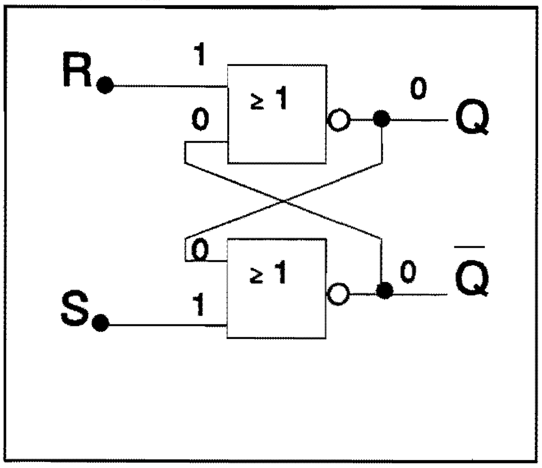
S0 = Wait
S1 = DropCup
S2 = DispDrink

Input condition				
Cur. state	Coin = False or Choice = None	Coin = True & Choice = Up	Coin = True & Choice = Coke	Coin = True & Choice = Orange
SO	S0; Off, Off	S1; On, Off	S1; On, Off	S1; On, Off
S1	-; -, -	S2; Off, Up	S2; Off, Coke	S2; Off, Orange
S2	S0; Off, Off	S0; Off, Off	S0; Off, Off	S0; Off, Off

Note that we have not specified what should happen in state S1 for the first input condition. According to the specification, this combination can never arise; this means that the ASM can never enter S1 as long as Coin = False or Choice = None. This degree of freedom could be used to obtain a simpler implementation. Also notice that the other input conditions in reality are less complex than shown here: input Coin could also have the value False.

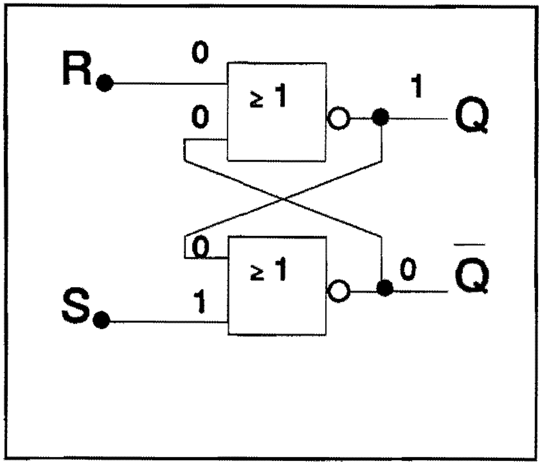
.1.

a. S = R = 1



The output of a NOR-gate is 0 when one of the inputs is a one. For the situation S = R = 1 holds that both flip-flop outputs are 0. Accordingly these outputs are not (as usually) complementary.

b. S goes to 0



The above situation is not stable. If one of the inputs (here: S) goes to 0, the corresponding output goes to one, and one of the stable states (here: set-state) are entered.

c. S and R goes to 0 simultaneously
If both S and R become 0 at the same time, both outputs become 1. The 1s are fed back to the inputs, hence the outputs again become 0.... and so on. The flip-flop would oscillate. This situation only arises, however, if both gate delays are identical. Because this never happens in a practical situation, the flip-flop will ultimately reach either the set or reset state. One cannot foresee in which state the system settles.

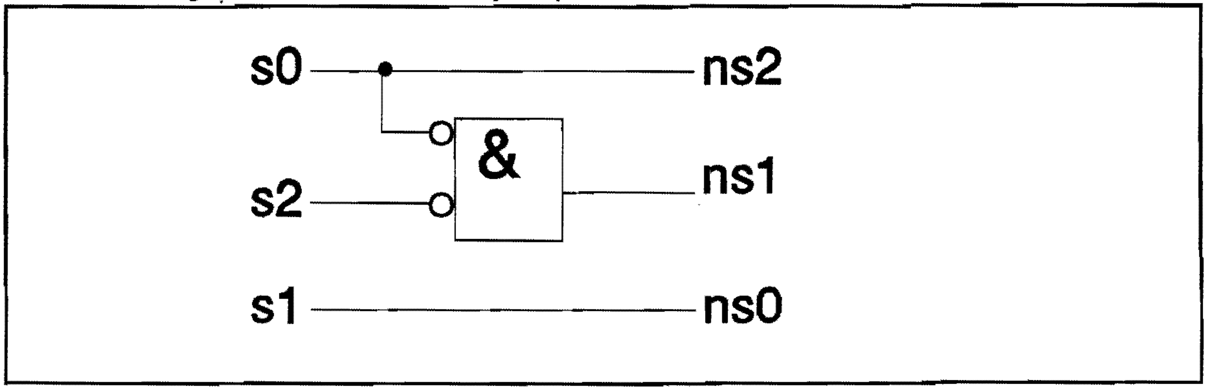
7.2. There are many Gray codes with 10 code-words. Below we show 2 of them:

1 1 1 1	0 0 0 0
1 1 1 0	1 0 0 0
1 1 0 0	1 0 0 1
1 0 0 0	0 0 0 1
1 0 0 1	0 0 1 1
0 0 0 1	0 0 1 0
0 1 0 1	0 1 1 0
0 1 0 0	0 1 1 1
0 1 1 0	0 1 0 1
0 1 1 1	0 1 0 0

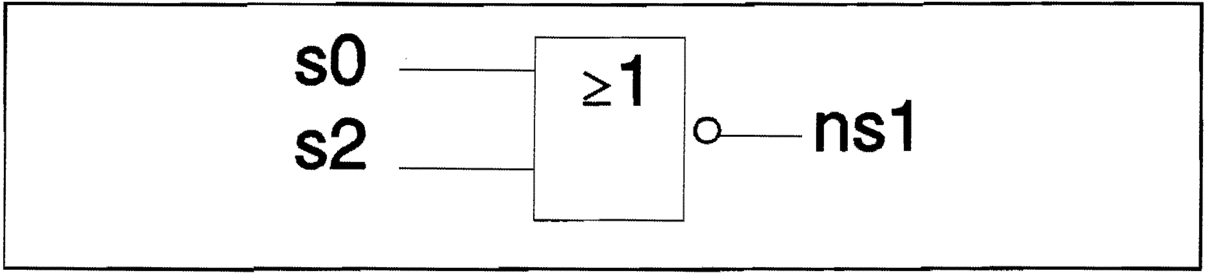
The state table is given in Fig. 7.20 of the survey. Because there are no inputs or input conditions, this table can also be interpreted as a truth table for the Next State function. The table directly gives the boolean expressions for ns_2, ns_1 and ns_0. The unused state codes lead to don't cares:

$$
\begin{aligned}
ns_2 &= s_2's_1s_0 + s_2s_1's_0 + \underline{s_2's_1's_0} + \underline{s_2s_1s_0} = s_0 \\
ns_1 &= s_2's_1's_0' + s_2's_1s_0' = s_2's_0' \\
ns_0 &= s_2's_1s_0' + s_2's_1s_0 + \underline{s_2s_1s_0}' + \underline{s_2s_1s_0} = s_1
\end{aligned}
$$

The circuit diagram hence becomes very simple:



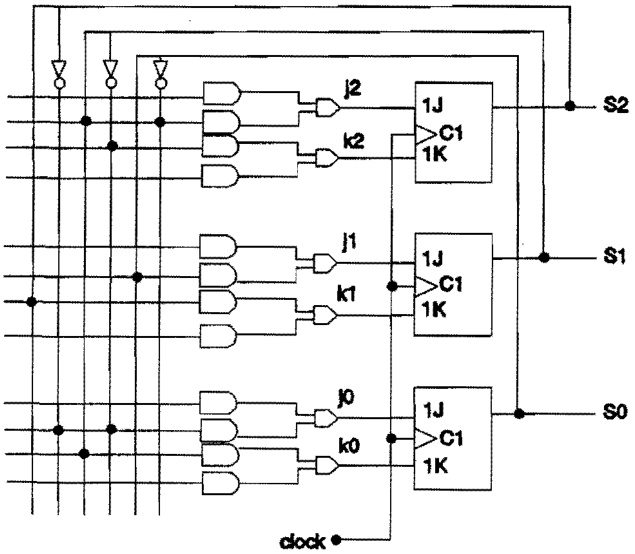
The and-gate can be replaced by a nor-gate:



7.4. Before we draw the circuit diagram, we derive the functions for the j- and k-inputs of the flip-flops from the exitation table in Fig. 7.22 of the survey.
By making good use of the don't cares we obtain:

$j_2 = s_1 s_0'$ $\quad$ $k_2 = s_1'$
$j_1 = s_0$ $\quad$ $k_1 = s_2$
$j_0 = s_2' s_1'$ $\quad$ $k_0 = s_1$

Below a circuit diagram is shown. A PAL is used with two product terms per sumterm output. This will do for our purpose:



7.5. Below we have combined the state table and the table for the exitation function. The table itself is self-explanatory:

s_2	s_1	s_0	ns_2	ns_1	ns_0	j_2	j_1	j_0	k_2	k_1	k_0
0	0	0	0	1	0	0	1	0	-	-	-
0	1	0	0	1	1	0	-	1	-	0	-
0	1	1	1	0	1	1	-	-	-	1	0
1	0	1	1	0	0	-	0	-	0	-	1
1	0	0	0	0	0	-	0	0	1	-	-

Regarding output no_0 we remark that the determination of the exitation function is not that simple. This is because no direct information is available about the current output value. This information could of course be derived from the state machine behaviour. In this specific case, each state has only one current output value; for that reason there is no problem. In the general case also input values play a role in determining the current and next output values.

6. We shall not give a complete worked-out solution for this problem; we skip the state diagram or ASM chart, and also the symbolic state table.

We begin with the coding step, choosing binary code for the state and output bits. The coded state table is given below; for a modulo-8 counter we remark that 3 state bits are needed, realized as 3 flip-flops. The Next State part of the table has 2 columns, one for each value of the clear input.

	clear = 0	clear = 1
$s_2\ s_1\ s_0$	$ns_2\ ns_1\ ns_0$	$ns_2\ ns_1\ ns_0$
0 0 0	0 0 1	0 0 0
0 0 1	0 1 0	0 0 0
0 1 0	0 1 1	0 0 0
0 1 1	1 0 0	0 0 0
1 0 0	1 0 1	0 0 0
1 0 1	1 1 0	0 0 0
1 1 0	1 1 1	0 0 0
1 1 1	0 0 0	0 0 0

For each flip-flop type we shall derive an exitation table, all combined in the table below. The D-flip-flops are straight-forward; the d-inputs are identical to the corresponding ns-values. The other flip-flops are more complex.

cl	$s_2\ s_1\ s_0$	$ns_2\ ns_1\ ns_0$	$d_2\ d_1\ d_0$	$j_2\ j_1\ j_0$	$k_2\ k_1\ k_0$	$t_2\ t_1\ t_0$
0	0 0 0	0 0 1	0 0 1	0 0 1	- - -	0 0 1
0	0 0 1	0 1 0	0 1 0	0 1 -	- - 1	0 1 1
0	0 1 0	0 1 1	0 1 1	0 - 1	- 0 -	0 0 1
0	0 1 1	1 0 0	1 0 0	1 - -	- 1 1	1 1 1
0	1 0 0	1 0 1	1 0 1	- 0 1	0 - -	0 0 1
0	1 0 1	1 1 0	1 1 0	- 1 -	0 - 1	0 1 1
0	1 1 0	1 1 1	1 1 1	- - 1	0 0 -	0 0 1
0	1 1 1	0 0 0	0 0 0	- - -	1 1 1	1 1 1
1	0 0 0	0 0 0	0 0 0	0 0 0	- - -	0 0 0
1	0 0 1	0 0 0	0 0 0	0 0 -	- - 1	0 0 1
1	0 1 0	0 0 0	0 0 0	0 - 0	- 1 -	0 1 0
1	0 1 1	0 0 0	0 0 0	0 - -	- 1 1	0 1 1
1	1 0 0	0 0 0	0 0 0	- 0 0	1 - -	1 0 0
1	1 0 1	0 0 0	0 0 0	- 0 -	1 - 1	1 0 1
1	1 1 0	0 0 0	0 0 0	- - 0	1 1 -	1 1 0
1	1 1 1	0 0 0	0 0 0	- - -	1 1 1	1 1 1

The following minimized exitation functions can be derived from the table:

$d_2 = cl' \cdot s_2 s_0' + cl' \cdot s_2 s_1' + cl' \cdot s_2' s_1 s_0 = cl' \cdot (s_2 \oplus s_1 s_0)$ $\quad d_1 = cl' \cdot (s_1 \oplus s_0)$ $\quad d_0 = cl' \cdot s_0'$

$j_2 = cl' \cdot s_1 s_0$ $\quad j_1 = cl' \cdot s_0$ $\quad j_0 = cl'$

$k_2 = s_1 s_0 + cl$ $\quad k_1 = s_0 + cl$ $\quad k_0 = 1$

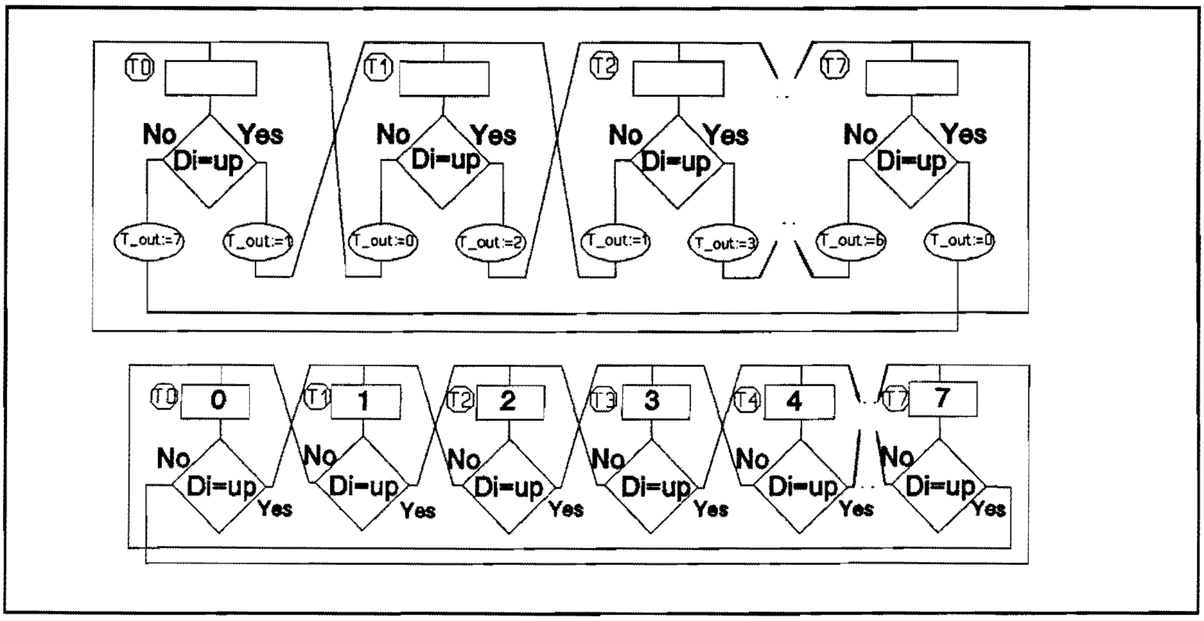
$t_2 = cl \cdot s_2 + cl' \cdot s_1 s_0$ $\quad t_1 = cl \cdot s_1 + cl' \cdot s_0$ $\quad t_0 = cl' + s_0$

We skip the circuit diagram; the functions are self-explanatory. We remark that the complexity of the solutions for the different flip-flop types do not differ significantly. Only for bit 0 we would clearly choose a jk-flipflop; here no gates would be needed.

7.7.

a. Below both (a part of) the ASM-chart and its simplified counterpart has been drawn. The states not drawn are similar.

Remark that the ASM-chart will be passed through to the left or to the right depending on the Direction input.



b. The states have been named T0 - T7.
The related output values are labeled 0 - 7. The state table now looks as follows:

	Direction	
Cur. state	Up	Down
T0	T1,1	T7,7
T1	T2,2	T0,0
T2	T3,3	T1,1
T3	T4,4	T2,2
T4	T5,5	T3,3
T5	T6,6	T4,4
T6	T7,7	T5,5
T7	T0,0	T6,6

Next state, Next Output

We use the same Gray code for the output code and the state code.
We choose Up = 1 and Down = 0 for the Direction code.

The following exitation table is valid for a certain Gray code:

Di	s_2	s_1	s_0	ns_2	ns_1	ns_0	t_2	t_1	t_0
0	0	0	0	1	0	0	1	0	0
0	0	0	1	0	0	0	0	0	1
0	0	1	1	0	0	1	0	1	0
0	0	1	0	0	1	1	0	0	1
0	1	1	0	0	1	0	1	0	0
0	1	1	1	1	1	0	0	0	1
0	1	0	1	1	1	1	0	1	0
0	1	0	0	1	0	1	0	0	1

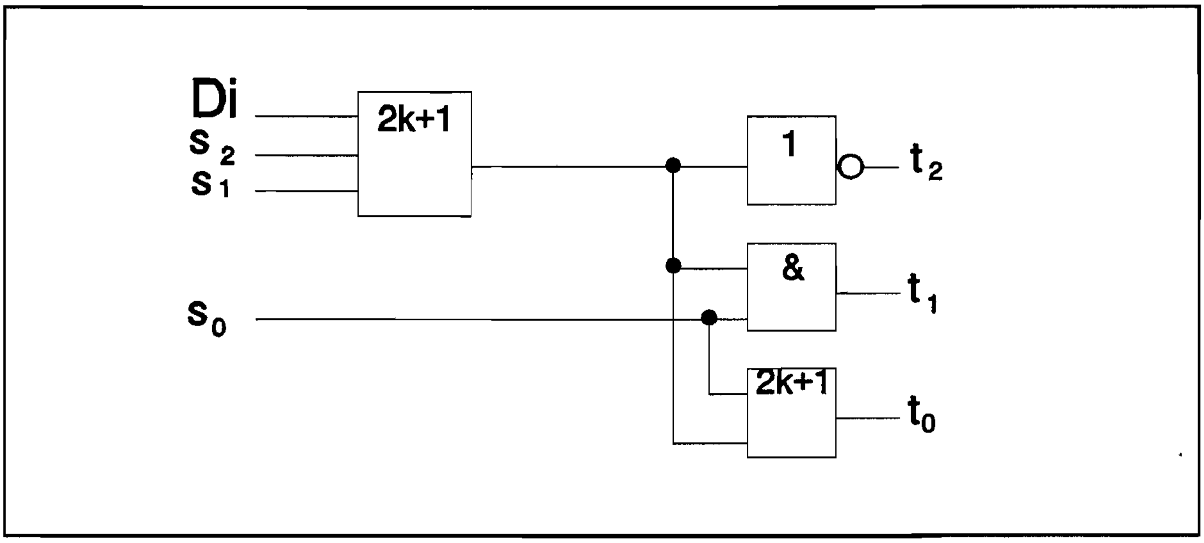
Di	s_2	s_1	s_0	ns_2	ns_1	ns_0	t_2	t_1	t_0
1	0	0	0	0	0	1	0	0	1
1	0	0	1	0	1	1	0	1	0
1	0	1	1	0	1	0	0	0	1
1	0	1	0	1	1	0	1	0	0
1	1	1	0	1	1	1	0	0	1
1	1	1	1	1	0	1	0	1	0
1	1	0	1	1	0	0	0	0	1
1	1	0	0	0	0	0	1	0	0

We determine the following expressions for the toggle flip-flop exitation functions:

$t_2 = (Di \oplus s_2 \oplus s_1)'$
$t_1 = s_0\,(Di \oplus s_2 \oplus s_1)$
$t_1 = Di \oplus s_2 \oplus s_1 \oplus s_0$

We easily see that we get exclusive-or functions, because the value of the xor-expressions in question are 1 for any combinations of its variables of which an odd number has the value 1.

The circuit diagram is given below:


Di
S 2
S 1
2k+1
1
t2
&
t1
S0
2k+1
t0