A DIGITAL SYSTEM DÉSIGN LANGUAGE

This thesis is submitted to the Department of Computing and Control at the Imperial College of Science and Technology, University of London, in fulfillment of the requirements for the degree of Doctor of Philosophy.

B. Shahdad, July 1974

ABSTRACT

Digital systems may be viewed at different descriptive levels: at the higher levels the designer is mainly concerned with the behaviour of the system; at the lower levels with the behaviour of its components and the way they are connected, i.e. the system structure.

The design of a system is thought of as involving several steps, each of which is concerned with the transfer of the system description from a particular level to a neighbouring one.

Among the design media being employed are natural languages, block diagrams, logical equations, and combinations of these. This thesis introduces a high level language to serve as a single medium throughout the design process. This approach, as well as providing a formal medium of communication, enables us to develop a processor for the Language which can simulate the behaviour of the system at each design step, before it is implemented in hardware. With regard to this application, the new problems in the implementation of the Language, as compared with conventional programming languages, are also investigated.

ACKNOWLEDGEMENTS

The work reported here was carried out under the supervision of Professor D.J. Howarth, to whom I am very grateful for his encouragement throughout the project.

The work was begun at the Institute of Computer Science in February 1972, following the closure of which, in September 1973, it was continued at the Imperial College of Science and Technology. The project was supported by a CENTO scholarship made available to the author through the efficient service of the British Council.

I would like to express my gratitude to Mr. I.H. Gould for constructive suggestions made in many discussions with Professor Howarth and the author. Thanks are also due to Mr. D.R. Brough for his comments on the implementation of the Language, and to Mr. C.T. Burton for his help in editing the manuscript.

The author has been on leave from Arya-Mehr University, Teheran, Iran.

CONTENTS

Abstract		2.
Acknowledgements		3.
Contents		4.
Chapter I	Introduction	9.
1.	The Digital Description System	
2.	Applications	
3.	The Process of Design	
4.	Levels of Description	
5.	The Requirements of the Language	
6.	General Considerations	
7.	Plan of the Thesis	
Chapter II	The Level of Logic Elements	18.
1.	Introduction	
2.	Boolean and Integer Constants	
3.	Primitive Elements	
4.	Graphic Description	
5.	Symbolic Description	
6.	<u>same</u> and <u>delay</u> Elements	
7.	Connectivity	
8.	Networks	
9.	Symbolic Description of a <u>delay</u>	
10.	Sequential and Parallel Operations	
11.	Ordering the Operations	
12.	Terminal Registers	

Chapter III	The Register Transfer Level	44.
1.	Introduction	
2.	Representation of Boolean Constants	
3.	The <u>unknown</u> Value	
4.	Arrays	
5.	Constants	
6.	Array Operations	
7.	Expressions	
8.	RT-operations	
Chapter IV	The Control Level	64.
1.	Introduction	
2.	Parallel Networks	
3.	Sequential Networks	
4.	Controlled Networks	
5.	General Networks	
6.	Systems	
7.	Implementation of General Networks	
8.	Higher Level Control Structures	
9.	An Example of Design	
Chapter V	Miscellaneous Extensions	90.
1.	Introduction	
2.	Logic Elements with Time Delay	
3.	System-defined Elements	
4.	Element Definition	
5.	Simulation Facilities	
6.	Input/Output Facilities	
7.	An Overall View of a Description	

Chapter VI The Implementation 111.

1. Introduction
2. The Choice of the Implementation Language
3. Major Strategies
4. Linking the Modules
5. Declaration of Modules
6. Referencing the Modules
7. Determination of the Base Address
8. The Virtual Stores
9. Declarations
10. The System Body
11. RT-operations
12. Considerations Concerning the delay Element
13. Delayed Boolean Operators
14. Element Designators
15. The for Structure
16. The Implementation Work

Chapter VII Concluding Remarks 129.

1. An Overall View
2. Other Design Languages
3. Use of Programming Languages
4. Formal Description
5. Simulation
6. Scheduled Event Simulators
7. Design Automation

Appendix A Description of a Computer 137.

1. Introduction
2. General Description
3. Effective Address Generation
4. Instructions
5. The System
6. Conclusion

Appendix B The Implementation Language 152.

1. Introduction
2. General Description
3. The BCL Sequencing Mechanism
4. Format Handling
5. Repeated Elements
6. Effectiveness for Semantic Analysis
7. Representation of Data Structures
8. The Scope of Variables
9. Concluding Remarks

Appendix C Definition of the Syntax 170.

1. Constants
2. Variables
3. Expressions
4. RT-operations
5. Sequential and Parallel Networks
6. The <u>if</u> Structure
7. The <u>for</u> Structure
8. The <u>case</u> Structure
9. Loop Structures
10. Miscellaneous Structures

11. Systems

12. Declarations

13. User-defined Elements

References 179.

CHAPTER I

Introduction

Digital systems serve for a variety of purposes: the field of computer systems alone gives a great importance to the study of their design.

The designers of digital systems have long been in search of suitable means of presenting the methods and products of their designs. English descriptions, lacking as they do precision, concision, and clarity, were soon replaced by block diagrams, which have the advantage of a certain correspondence to the physical configuration of the system. An alternative language was suggested by the application of Boolean algebra to network-simplification, and logical equations have been used, both alone, and in combination with block diagrams. The relative merits of the two methods have been the subject of several discussions and papers [9,30,46].

The use of flowcharts, which are capable of describing the sequencing and operation of systems, has been suggested and design problems have been presented for which this method is effective [38].

In general, the designer might use a combination of these or other languages in his design: a state diagram might present the initial statement of the problem; the notation of logical equations might serve to observe certain design restrictions and simplify the system; the final solution might be presented in the form of a block diagram.

It is desirable to abstract from these languages the features which are pertinent to the design process, to obtain a single language which could serve as the design medium throughout.

This thesis is concerned with the investigation of the required features of such a language. In recent years several attempts in various directions have been made to evaluate the effectiveness of this approach, e.g. [39,14,3]. Due to their diversity - embracing the fields of digital system design, language design, translator writing, and simulation - and their lack of established terminology, we review them, in our final chapter, in the context of the terminology and conceptual framework of this thesis.

In this chapter we first expand the idea of a high level digital system design language to define what is called the Digital Description System; the purposes for which the Description System could serve are considered, and the requirements of the corresponding Description Language are set out in relation to the multi-level structure of digital systems, and the process of their design; finally we mention characteristics of digital computer systems affecting simulation.

1. The Digital Description System

The Description System is composed of three components: the Description Language, which is the notation in which the user describes his design; a simulator which simulates the behaviour of the system described; and a compiler which can be regarded as an interface between the first two components. The compiler accepts the source description and produces an object description which is acceptable to an object machine. The simulator among other things interprets the object code.

2. Applications

The Description System could serve for three kinds of purposes:

a) It could provide a medium of communication in the same way that programming languages have been used to express computational processes. Such a description is formal, concise, precise, and expressive.

b) The simulator could assist the designer in ensuring the proper working of the system, in investigating its behaviour, and in improving the design before it takes a hardware form. Once the design has been approved, information for assembling the hardware could be derived from the description, with obvious saving in design time and cost. Also effects of mulfunctions in any part of the system could be simulated and investigated.

c) The Description Language could conceptually help the designer in his creative work by offering facilities which encourage him to take a systematic approach in his design; this is further discussed in ensuing sections.

3. The Process of Design

In designing the Language we have had in mind the design process, known as the 'stepwise refinement method', which recently has been considered in the design of programs [58].

According to this process the designer begins with the problem statement, i.e. the specification of the 'component' to be designed. The design task involves several steps. In each step the design of a component is refined, that is, the solution is expressed in terms of the design of sub-components of a lower degree of complexity. This is achieved by decomposing the complex operations into simpler ones, and refining the representation of data into more primitive forms. This process is continued, directing progress toward a level whose 'primitive components', i.e. the components forming the solution, are available to the designer, thus accomplishing the design task. The primitive components of this final level are also referred to as 'primitive elements'.

A component designed in this manner might become part of a more complex system.

3.1 The Design Tree

As pointed out in [35] one can imagine a design tree whose root, P_0, represents the initial statement of the problem; the leaves of this tree correspond to possible solutions to the problem; each level of the tree contains all the possible refinements of the previous level.

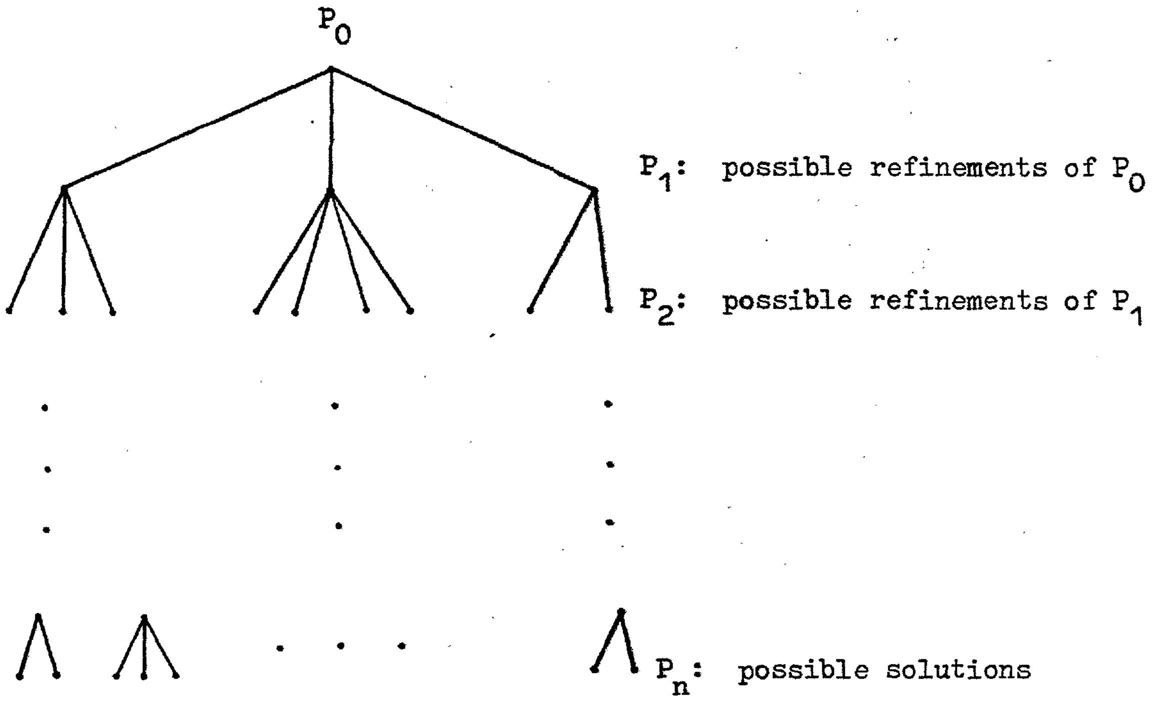

P_0
P_1: possible refinements of P_0
P_2: possible refinements of P_1
P_n: possible solutions

Figure 1. The design tree

In each refinement step the designer, on the basis of certain design criteria, makes design decisions which take him to a node belonging to the next design level.

It may well happen that after reaching a level the designer wishes to reconsider the decisions made at a previous level, and to select an alternative design route. Similarly, the designer might make a choice on the basis of looking ahead one or two levels. An example of trade-offs between various design decisions in the design of a multiplier, leading to several solutions, can be seen in [6].

3.2 Features of this Method of Design

This approach toward design has the following features:

Firstly, the correctness of the design can be ensured at each level so that ideally the final design product is free of logical errors.

Secondly, the design product is accompanied by a design history which shows the decisions made and the reasons for them. While the solution at the final level is suitable for implementation, it obscures the architecture of the design due to too much detail. Intermediate solutions, which are presented at higher levels, aid comprehension of the way the final product has been developed.

Finally, from the documentation viewpoint, if a change is made to the system at a certain level, only the descriptions at lower levels are affected. The nature of the change might be a physical modification or reconsideration of a design decision. Since the design is well documented, modifications and maintenance should prove to be easier.

4. Levels of Description

As mentioned earlier a digital system may be described at different levels. At each level the behaviour of the system is defined by the behaviour of its components and the way they are connected, that is the 'system structure'. The lowest level considered here is the one whose primitive components are logic elements, and is therefore referred to as the 'Level of Logic Elements', at which each logic element produces an output signal after performing a certain logical operation on its input signals, possibly (depending on the type of element) with a time delay.

The next higher level is the 'Register Transfer Level', at which a digital system is imagined to be composed of registers connected through blocks of logic elements. Each block performs an operation on the contents of one or more input registers, and places the result in output registers, which may include some of the inputs.

The next higher level is the 'Control Level' where the order in which the operations take place is expressed; an operation may be invoked at a particular step in a sequence of steps, or at a certain time, or in general when certain conditions are satisfied. Such conditions may well be a function of the previous states of the system.

The levels stated above are the important ones. The descriptive levels of digital systems are not rigid or restricted to these; thus it might prove convenient to extract a sub-level from a level, or extend the range of a level. For example, the Control level might be extended to include the instruction level, at which a machine is described in terms of the changes in its states as the result of the execution of each instruction.

At the highest descriptive level a digital system is in general composed of a small number of components, each with a high degree of complexity. As we move downwards, the number of components increases while the degree of complexity of each component decreases. The invariant of this process is the behaviour of the aggregation of the components - i.e. the system.

5. The Requirements of the Language

With regard to the purposes for which the Language could serve, the process of design mentioned earlier, and the descriptive levels of digital systems, we state the following requirements for the Description Language.

a) The same language should be used throughout the design so that the designer need not be obliged to learn several notations (possibly corresponding to different descriptive levels), and so that components of the system described at different levels can interact with each other.

b) In order for the Language to be used throughout the design, it should have a multi-level structure, and be flexible to the designer's choice of levels.

It should be able to suppress details so that, at a high level, the designer can investigate the overall interaction of a component with the rest of the system, or ignore structural details. Similarly, if a component has already been designed in detail, one should be able to refer to it at a higher level without going into its structural details, and by merely complying with the requirements of the interface.

c) The concepts on which the Language is based should provide for the description of the complex behaviour of components at a high level, where only a few components are involved, and at the same time for the description of connectivity at a low level where the system is composed of numerous components.

d) The Language should be equally suitable whether the design approach is top-down, bottom-up, or some combination of the two. From this viewpoint its main concern should be to cater for the design of components in terms of sub-components.

e) As stated in [57] the required power and flexibility should derive from a unified simplicity rather than the inclusion of integrated facilities. This should be combined with the Language ability for multi-level description in such a way as to offer the designer a conceptual aid for progressing toward the implementation level, presenting the solution in terms of the pre-defined components which he is likely to have at his disposal.

f) An ease of expression should be achieved by building into the Description System the abstract behaviour of digital systems. Once the user feels an ease of expression, one of our main objectives, that of creating a medium of communication and a way of documentation, is achieved. For when it is easy to describe, it is easy to read; the description is then self-documented and self-expressive.

g) One of the important requirements of the Language is to allow the designer to define his own operators and use them in the same manner as the standard operators so that the description is homogeneous. This enables the user to describe his own technology and design his components in terms of this technology.

h) Finally, the Language should lend itself to implementation so that a simulator can be developed to ensure the proper working of the system at every level, and therefore to raise the designer's degree of confidence in his design.

6. General Considerations

The 'model' represented in the computer is in general an approximation to the real digital system. The words, model and system, are used in contrast to each other when one is concerned with those features of the system which are abstracted to obtain the model; otherwise the word, system, refers to both.

Using a digital computer for the purpose of simulation implies that the continuous advance of the 'Real Time' in which the digital system runs should be approximated by discrete changes in the variable representing it in the model. This variable is called the 'Simulated Time', or '*time*'.

Having only one processor available, the effect of parallelism is created by making available to the receiving component the result of the propagation of parallel signals only after they have all been propagated.

As the simulation process continues the variable know as the 'Computation Time', which corresponds to the computer system used for the purpose of simulation, advances. The following figure shows the correspondence between different systems and Times involved in the process.

$$\begin{tikzcd}
\text{digital system} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \text{Real Time} \arrow[d, leftrightarrow] \\
\text{model} \arrow[r, leftrightarrow] & \text{Simulated Time} \\
\text{computer system} \arrow[r, leftrightarrow] & \text{Computation Time}
\end{tikzcd}$$

Figure 2. Systems and Times

7. Plan of the Thesis

Chapters II, III, and IV of this thesis are concerned, respectively, with the Levels of Logic Elements, Register Transfer, and Control. Chapter V is mainly devoted to the extension of the basic set of operators of the Language, and ways of composition and decomposition of systems in this light. In Chapter VI problems involved in the implementation of the Description System are considered; BCL - the implementation language - is described in Appendix B. Finally, Appendix C gives the syntactic definition of our Language, and Appendix A illustrates its application in presenting a user-oriented view of a small computer.

CHAPTER II

The Level of Logic Elements

1. Introduction

It is the purpose of this chapter to define the notion of a network, and to study such networks at the Level of Logic Elements. Logic designers often use graphic descriptions to present their design, and we begin by establishing a formal correspondence between this type of description and symbolic notation, since the latter is a more suitable form for computer input, and for representing digital systems at other levels of design.

The notation is then used to introduce and study certain categories of networks, which are regarded here as basic constituent elements of digital systems. Detailed attention is paid to the representation and behaviour of delay elements, from which a suitable notation is developed.

At the same time a theoretical foundation is set up, to form a basis for identifying the requirements of a simulator for the behaviour of digital systems described at this level.

2. Boolean and Integer Constants

A Boolean constant is a member of the set

$$B = \{\underline{\text{true}}, \underline{\text{false}}\}$$

and an integer constant is a member of the set

$$Z = \{0, -1, 1, -2, 2, \ldots\ldots\}$$

The constant, unknown, is used to indicate an undefined value for a variable.

3. Primitive Elements

The notion of a 'primitive element' corresponds to a function from the set of its 'input' values into the set of its 'output' values; the element has an arbitrary number of inputs (n), and a single output. The functional form is:

$$f: \quad I \rightarrow O$$

where $$I = i_1 \times i_2 \times \ldots \times i_n, \quad n \geq 1$$

and each i_j is either B or Z; and similarly, O is either B or Z.

'Terminal' is a general word for referring to an input or output. A terminal may take values from an appropriate set of constants. In this respect, we can talk about 'terminal variables'. The value of such a variable is often called a signal; therefore, a signal is either of 'type' Boolean or integer.

3.1 Naming

Terminals may be given names for purpose of reference. Two identical names indicate that their corresponding values are always the same; therefore distinct terminals should be given distinct names.

The whole element may also be given a name, in which case the name is written on the left hand side of its output identifier, separated by a colon (:).

4. Graphic Description

Figure 1 shows graphic description of a primitive element. There are arrows on input and output, pointing towards and out from the centre. If f is a standard function, such as $\geq$, $\sim$, +, etc., the corresponding symbol is entered in the circle; otherwise the function name is written. Such a composite function should be described in terms of standard functions.

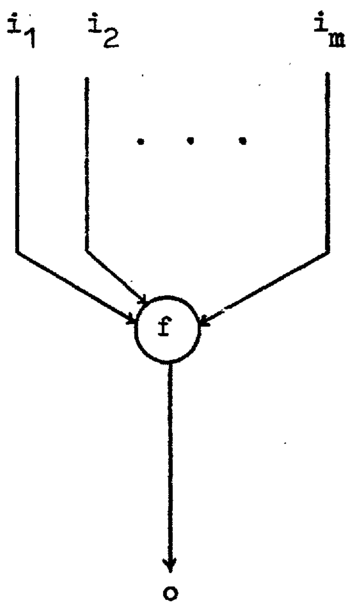


Figure 1. General form of a primitive element

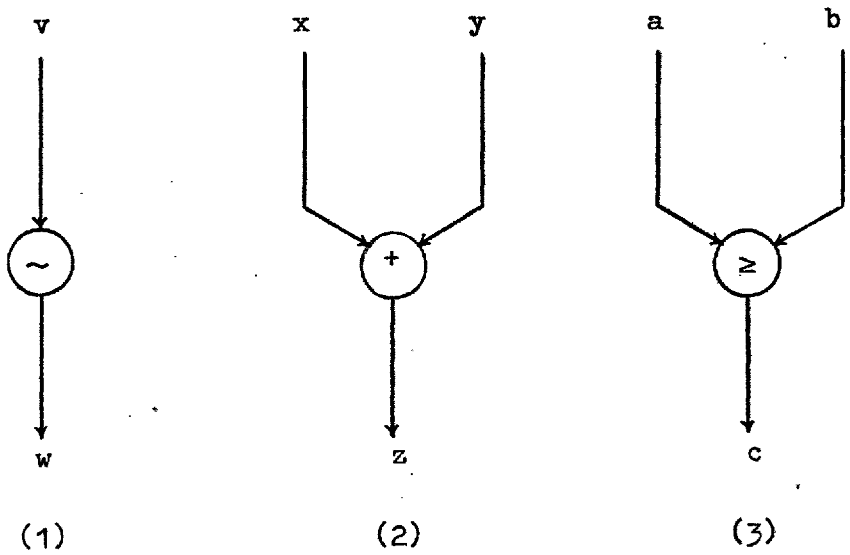


Figure 2. Examples of primitive elements

5. Symbolic Description

The symbolic representation of a primitive element is of the form:

$$o = f(i_1, i_2, \ldots, i_n);$$

The above form is called a statement. Examples are:

1) $w = \sim v;$

2) $z = +(x,y);$

3) $c = \geq(a,b);$

corresponding to the graphic descriptions in Figure 2.

6. Same and delay elements

The statement

(1) $b =$ same $a;$

is a symbolic description of the function same, whose output value is always the same as its input value, i.e. it is the identity function. A simpler way to write statement (1) is:

$$a = b;$$

The function same may be used to change the name of a signal, in this case from b to a.

Figure 3 is a description of the delay element. The two input variables are not independent; in fact,

(a) $x = f(t)$

for the output, we have

(b) $y = f(t-n)$

where $n \geq 0$ is an integer constant. t is usually referred to as time. Since $x = f(t)$, t need not explicitly be shown.

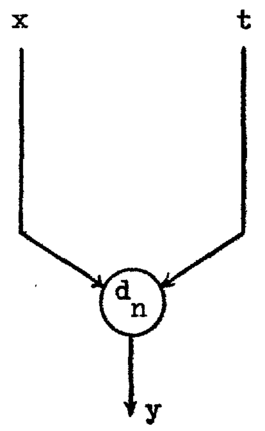


Figure 3. The delay element

There are two special cases in which delay behaves like same:

1) if $n = 0$

2) if x is a constant. In this case f is a function whose range is a constant, and therefore $y = x$.

If the function f is not known analytically, the tabulated form is available. Let tabulation points be

(c) $t_i = t_{i-1} + 1, \qquad t_0 = 0$

then

(d) $y_{i+n} = f(t_{i+n} - n) = f(t_i) = x_i$

Assuming that time is increasing according to (c), if a value y_i is to be released at $t = t_i$, a buffer of n locations is needed to store

$$x_j \quad \text{for all} \quad i-n \leq j \leq i-1$$

since according to (d)

$$y_i = x_{i-n}$$

Values enter and leave the buffer on a queue basis, i.e. first in, first out. All y_i, $0 \leq i < n$ are unknown. If after $t = t_i$ no value enters the buffer, those already in will be released at time points

$$t_j, \; i+1 \leq j \leq i+n$$

6.1 Functional Form of delay

In order to treat delay elements in the same manner as other primitive elements, a functional form is needed to represent them. The required function is:

$$y_i = x_{i-n}$$

This function is in a tabulated form. The symbolic representation of delay is discussed in Section 9.

6.2 Sub-classes of Primitive Elements

Three sub-classes of primitive elements are introduced here. The members of each sub-class accept a certain type of input signal and produce a certain type of output signal.

a) Boolean elements: and (&), or (v), not (~).

The terminal signals of these elements are of type Boolean.

b) Relational elements: lt (<), le (≤), eq (=), ne (≠), ge (≥), gt (>). The input signals are of type integer, and the output signal is of type Boolean.

c) Arithmetic elements: plus (+), minus (-), mult (*), div (/). The terminal signals are of type integer.

7. Connectivity

7.1 Graphic

In graphic description connectivity is indicated by joining lines representing input or output. If two lines are joined, the signals on both lines are the same (in either direction) up to the point where they reach a primitive element. Connection of two outputs as shown in Figure 4 is not 'valid'.

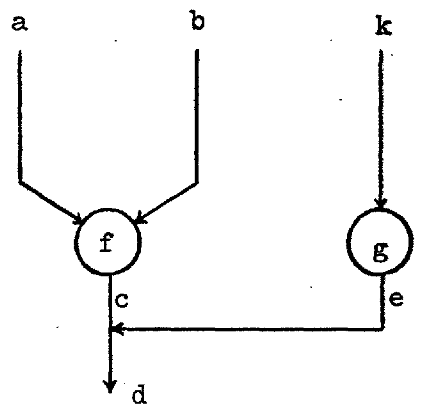


Figure 4. An invalid connection.

The reason such a connection is invalid is that: if c and e are not always the same, we should express what combination of the two exists at d. This is done through replacing the connectivity by a primitive element; otherwise the signal at d is 'ambiguous'. (An ambiguous signal is different from an unknown signal). If c and e are always the same, the network is 'redundant'.

7.2 Symbolic

Since distinct identifiers represent distinct signals, similar identifiers may safely be used to indicate connectivity. For example, the following is a symbolic description of Figure 5. Elements l_a and l_b are connected.

l_a: c = f(a,b);

l_b: d = g(c);

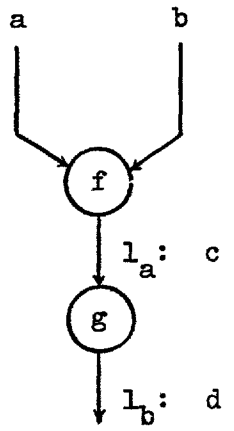


Figure 5.

Let L and R represent the left and right hand sides of a statement l. In terms of symbolic description, connectivities due to a set of statements are valid if there are no two statements, l_i and l_j $(i \neq j)$, such that

(a) $L_i = L_j$

If there exist a pair of statements satisfying (a) and also

(b) $R_i = R_j$

the network is redundant; otherwise, if (a) holds but not (b), an ambiguous signal will be produced.

8. Networks

A 'network' is a set of primitive elements with a set of valid connections over them.

8.1 Tree Networks

We define an input to be 'free' if it is not connected to an output.

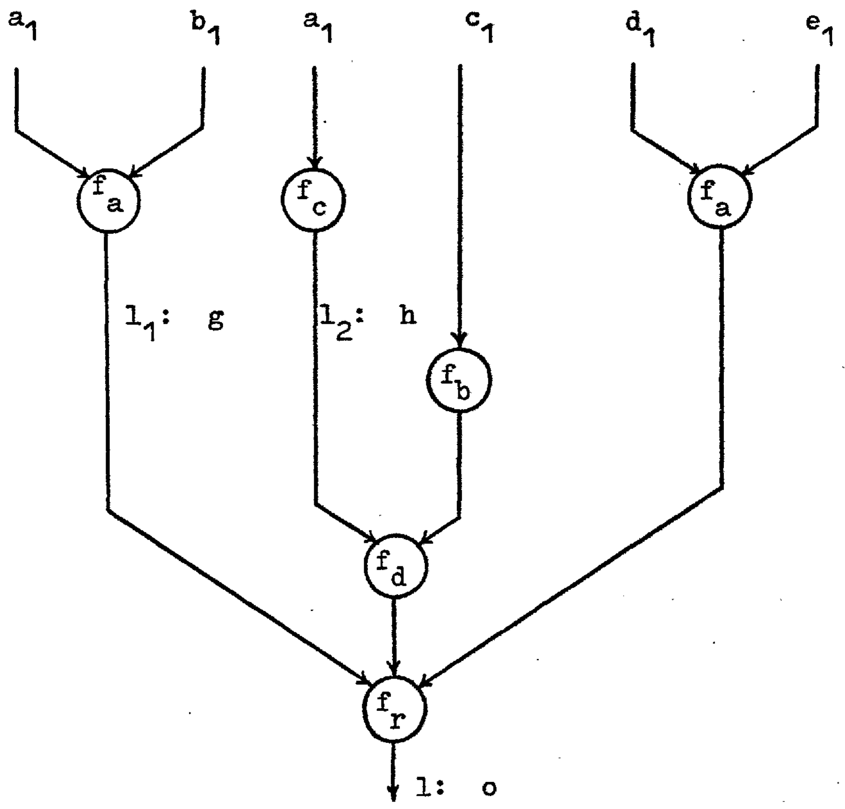
Similarly, an output is free if it is not connected to an input. The following is a recursive definition of a 'tree network': a 'tree network' is either

a) a primitive element, or

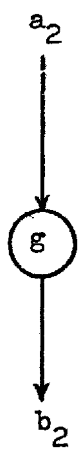
b) a primitive element with free output provided that either each of its inputs is free, or the only output connected to it is the output of a tree network.

The above definition covers both symbolic and graphic descriptions. It excludes all networks containing invalid connections.

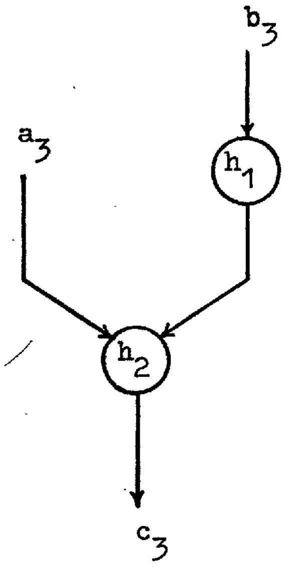
'Terminals' of a tree network are those which are free. The element from which the network output emanates is the 'root' of the tree. Since each tree has only one output, its root is unique. A tree is called by the name of its root, therefore it should be made clear which one is meant, when a reference is made.



(a)



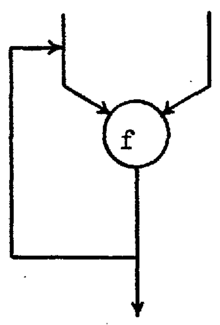
(b)



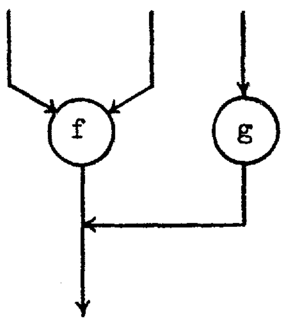
(c)

Figure 6. Tree networks

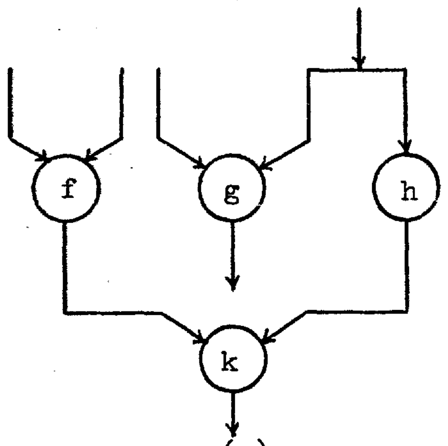
Figure 6 shows three examples of trees. Inputs of 1 in 6.a are a_1, b_1, c_1, d_1, e_1, and its output is o. The root of the tree is the element 1 and the function corresponding to the root is f_r.



(a)



(b) Figure 7.



(c)

Networks described in Figure 7 are not trees. (a) has no free output, (b) contains an invalid connection, and (c) has an extra element.

8.1.1 Behaviour of a Tree Network

The functional property of a tree network is called its behaviour. Let i_s, $1 \le s \le p$ be the inputs of a tree l. Let o be the output of l. We are looking for a function f such that:

$$o = f(i_1, i_2, \ldots , i_p);$$

Let f_r be the function corresponding to the element l. Let l_i, $1 \le i \le m$ be the trees connected to l, and g_i, $1 \le i \le m$ be the functions corresponding to these trees. Let k_{ij}, $1 \le j \le n_i$ be the set of inputs to the tree l_i; then,

a) if l is a primitive element, then

$o = f_r(i_1, i_2, \ldots , i_p)$; otherwise,

b) $o = f_r(g_1(k_{11}, k_{12}, \ldots , k_{1}n_{1}),$
$\ldots , g_i(k_{i1}, k_{i2}, \ldots , k_{ij}, \ldots , k_{in_i}), \ldots ,$
$g_m(k_{m1}, k_{m2}, \ldots , k_{mn_m}));$

Clearly, the functions g_i are determined in the same manner as f. Note that if an input a is free, it can always be assumed that it is the output of a tree network, say

a = <u>same</u> b;

As can be seen from the functional expansion, the behaviour of a tree network depends on the behaviour of its elements and the way they are connected, i.e. the tree 'structure'. Therefore, the behaviour of a tree is independent of the textual position of the statements representing that tree. Similarly, graphic transpositions do not affect the behaviour of a tree provided that the tree structure is retained. Two trees are said to be 'equivalent' if they behave the same way.

In the following example, the general functional form derived in this section is employed to determine the function corresponding to the tree l. From Figure 8 we have,

$m = 2,\ n_1 = 3,\ n_2 = 1,\ p = 4$

$f_r = \underline{\text{and}},\ g_1 = \underline{\text{or}},\ g_2 = \underline{\text{not}}$

$k_{11} = a,\ k_{12} = b,\ k_{13} = c,\ k_{21} = d$

hence,

$g = \underline{\text{and}}\ (\underline{\text{or}}(a,b,c),\ \underline{\text{not}}\ d);$

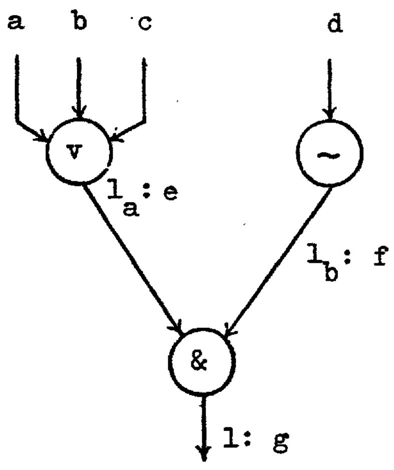


Figure 8.

8.1.2 Sub-trees

'Sub-trees' of a tree l are those whose output is connected to an input of the element l. Each sub-tree has its own terminals; these can be determined by treating it as a tree. In Figure 8, l_a and l_b, are sub-trees of l. The following relation exists between the inputs of a tree and the inputs of its sub-trees:

$$\bigcup_{\substack{1 \le i \le m \\ 1 \le j \le n_i}} \{k_{ij}\} = \bigcup_{1 \le s \le p} \{i_s\}$$

that is, the set of inputs of sub-trees of a tree, is the same as the set of inputs of the tree itself.

8.1.3 Well-formed Trees

Consider the element l described as:

$l\colon\ o = f(i_1, i_2, \ldots, i_n);$

A set of input signals to this element is identified by the ordered n-tuple

$$S = (s_1, s_2, \ldots, s_n)$$

In order for l to be able to operate, types of i_j and s_j for all $1 \leq j \leq n$ should match; they should be either both integer or both Boolean. The signal is then said to be 'correct'. If the tree structure is such that, given any correct set of input signals, each element of the tree receives correct signals, the tree is 'well-formed'.

8.1.4 Linear Description of a Tree

It was mentioned in Section 8.1.1 that the behaviour of a tree is independent of the textual position of statements forming its symbolic description. If the statements are arranged in a sequence such that every input of an element l is textually described before the description of l, the arrangement is said to be 'linear'. Clearly, if an input to an element is also a tree input, it need not be described further.

In general the linear description of a tree is not unique; furthermore each tree has at least one linear description, for if a network cannot be described linearly, there is at least one statement

$$l: \ o = f(i_1, i_2, \ldots, i_n);$$

with an input i_j such that i_j is neither a tree input, nor can be described before l. This implies that the description of i_j depends on the description of o, in which case the network does not have the tree property. For example, a linear description of the tree,

```
        h = or(f,a);
        d = not e;
        f = and(g,d);
        a = and(b,c);
```

is:

```
        a = and (b,c);
        d = not e;
        f = and(g,d);
        h = or(f,a);
```

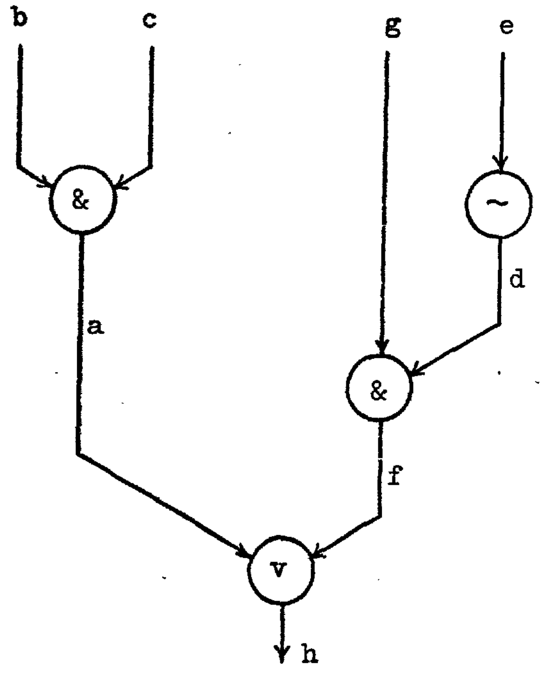
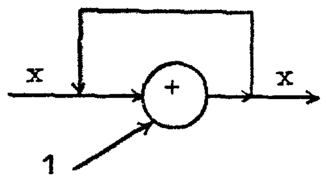


Figure 9.

The networks

```
        x = plus(1,x);
```



and

```
        b = not a;
        c = not b;
        a = c;
```

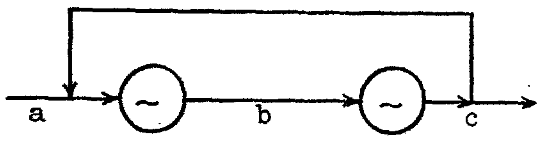


Figure 10.

cannot be described linearly. In the first network, description of x depends on itself, and in the second one, a and c depend on each other.

8.2 Composite Trees

A 'composite tree' is either

a) obtained from a tree by connecting at least one of its inputs to one or more of the elements of another simple tree, or

b) obtained from a tree by connecting at least one of its inputs to one or more of the elements of a composite tree.

Trees defined in Section 8.1 are referred to as 'simple trees' in contradistinction to composite trees. In general a composite tree has several outputs. The definition of the linear description of a composite tree is the same as that of a simple tree. Following is a graphic and linear description of a half-adder in the form of a composite tree.

```
c = and(a,b);
x = or(a,b);
s = and(x, not c);
```

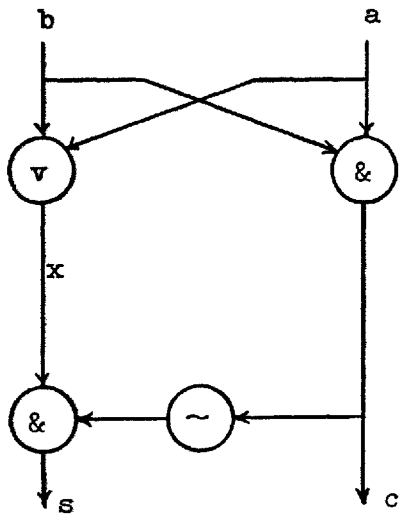


Figure 11. A composite tree

Like simple trees, each composite tree has at least one linear description.

8.3 Closed Trees

A 'closed tree' is either

a) obtained from a simple or composite tree by connecting at least one of its outputs to one or more of its inputs, or

b) obtained by validly connecting a closed tree to a simple, composite, or another closed tree.

The definition of a linear description is relaxed for closed trees, in that the closing terminals need not be described before they are referenced. If a closed tree has a closing terminal t, the symbolic description of that tree will include a network of some form equivalent to:

$$t = f(\ldots , t, \ldots)$$

As this form suggests, the working of the network is 'repetitive', in the sense that the output signals on closing terminals are fed back as input.

Trees defined in Sections 8.1 and 8.2 are referred to as 'open trees' in contradistinction to closed trees.

8.4 Networks

We can now give a more precise definition of a network: a network is a set of single/composite/closed trees described together. A linear description of a network is one in which each tree belonging to the network is described linearly. Networks more complex than a certain degree are referred to as 'digital systems'. There is no precise definition for the boundary whereafter a network is, or should be called a digital system.

9. Symbolic Description of a delay

The discussion of this topic has been deferred up to this point so that examples could be drawn from tree structures for its illustration and justification.

The functional form of a delay, i.e.

$$y_i = x_{i-n}$$

states that the output value of the delay is equal to its input value with a time lag. This suggests calling the terminals of the delay by the same name; in fact one can regard this name as the name of the delay itself. On the basis of this interpretation, we adopt a declarative form for introducing the delay elements in a network; for example,

```
delay cnt(1:16);
```

declares that cnt is a 16-bit delay element whose storage locations run from c(1) to c(16), with c(1) being at the front of the queue (the oldest member), and c(16) at the end of the queue (the youngest member).

Distinction between the input and output of a delay is made at the time the connectivity is established. If cnt appears on the left hand side of a statement, then the input of the delay is connected to the output of the right hand side tree; whereas if cnt is referenced in the right hand side, the output of the delay is meant.

As an example, the half-adder in Figure 11 is turned into a 16-bit serial counter [26]; cnt holds the count, and at every 16-bit time interval a true signal appears at x. Without going into the details of the working of this device, its linear description is given in Figure 12.

```
1-  delay cnt(1:16),b;
2-  c = or(x,b);
3-  d = and(c,cnt);
4-  b = d;
5-  cnt = and(or(c,cnt), not d);
```

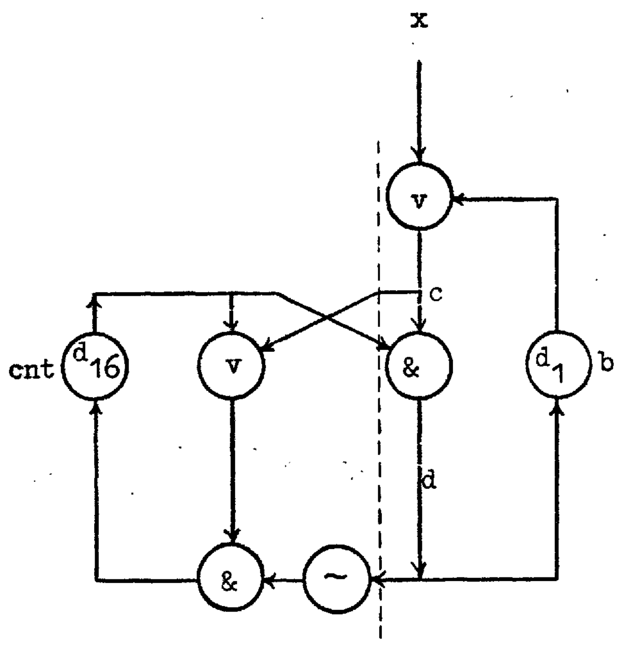


Figure 12. A serial counter

The above network is a closed tree with two closing terminals, namely cnt and b. Lines 2 to 4 describe the tree located to the right of the dotted line; the rest of the network is described in line 5; note that the name, cnt, at the left hand side of this statement refers to the input of the <u>delay</u>, whereas the same name at the right hand side refers to the output of cnt, with a 16-bit time delay with respect to the input.

The notation introduced here for the symbolic representation of the <u>delay</u> has, apart from a close correspondence to its functional form, three other important advantages discussed in the following sections.

9.1 Connection to other Elements

Connecting an element to a <u>delay</u> is quite simple; this is especially useful if one wishes to introduce the inherent delays of logic elements. For instance, an <u>and</u> gate which has a 1-bit propagation delay can be described as:

```
delay c;
c = and(a,b);
```

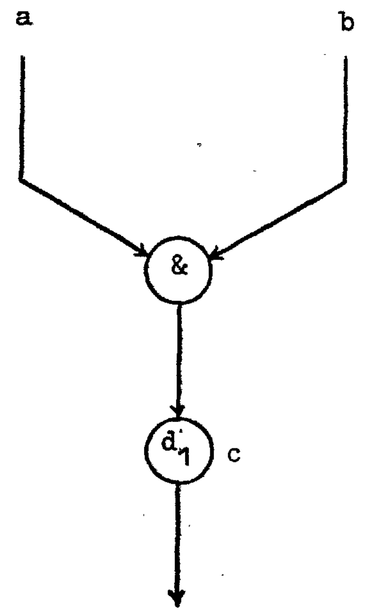


Figure 13. An *and* gate with a propagation delay

If the above tree were part of a network, the name c, which could only be referenced in the right hand side of the statements of the network, would correspond to the output of the *delay*, and thus the 1-bit propagation delay could easily be introduced. Since a 1-bit propagation delay is quite common, later (Chapter V) we shall see simpler ways of coping with this special case.

Another example of the connection between *delay* and other elements can be seen in line 5 of the example already given in Figure 12.

9.2 Initialization and Store Allocation

The second advantage concerns the fact that one has access to the members of the line (storage buffer associated with the *delay*). This could be helpful as it is frequently desired to regard a *delay* as a storage element during the initialization time. For example, the following segment initializes the first element of cnt to *true*, and the rest to *false*.

```
cnt(1) = true;
cnt(2:16) = false;
```

Finally, with regard to implementation, no matter what notation is adopted, one has to allocate an area of store, of appropriate size, to the delay line. In this respect, the notation given here treats the *delay* introductions almost in the same manner as declarations, and thus simplifies the implementation task.

10. Sequential and Parallel Operations

An output signal is produced as a result of primitive elements operating on their input signals, and thus propagating them. An operation is called by the name of the primitive element it corresponds to. We define the following relations between pairs of operations; by their nature they may also be referred to as meta-operators.

a) x is 'parallel' to y, denoted by $x \doteq y$

b) x is 'sequential' to y (x after y), denoted by $x \cdot\!> y$

These have the following properties:

I) if $x \cdot\!> y$ and $y \cdot\!> z$ then $x \cdot\!> z$

II) if neither $x \cdot\!> y$ nor $y \cdot\!> x$ then $x \doteq y$

III) if $x \doteq y$ and $y \doteq z$ then $x \doteq z$

IV) if $x \cdot\!> y$ and $y \doteq z$ then $x \cdot\!> z$

V) if $x \cdot\!> y$ and $x \doteq z$ then $z \cdot\!> y$

VI) if $x \doteq y$ then $y \doteq x$

VII) $x \doteq x$

Property VI is in fact a consequence of II.

The above axioms represent the nature of parallel and sequential operations, and their inter-relationship. Among these, axiom II has a special importance, for it ensures that one of the two relations, $\cdot\!>$ or $\doteq$, holds between any two objects belonging to a set of operations.

From the behaviour of a tree network it is understood that when propagating the signals the operation corresponding to the root of the tree should be carried out after all operations belonging to the sub-trees of that tree have been performed. Therefore, if x is the root of an open tree, and y is a sub-tree of x, we define

(1) $x \cdot\!> y$

As an example, consider the following tree:

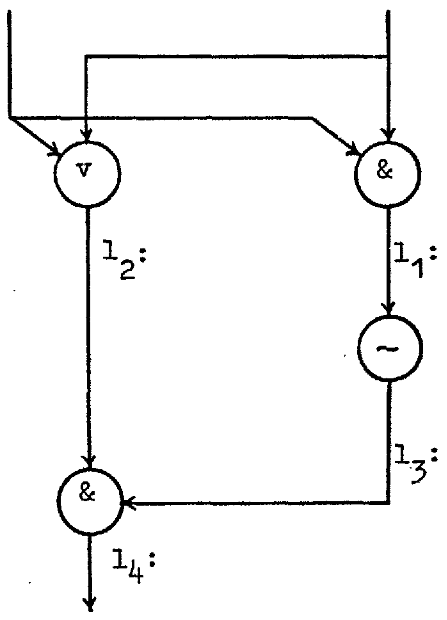


Figure 14.

According to the above definition, the following relations hold over the set of operations $A = \{l_1, l_2, l_3, l_4\}$:

a) $l_4 \cdot\!> l_2$

b) $l_4 \cdot\!> l_3$

c) $l_3 \cdot\!> l_1$

According to axiom II, since neither $l_2 \cdot\!> l_3$ nor $l_3 \cdot\!> l_2$, we have

d) $l_3 \doteq l_2$

This means that l_3 and l_2 are parallel. Axiom I and relations (b) and (c) result in

e) $l_4 \cdot\!> l_1$

Axiom V together with relations (c) and (d) results in

f) $l_2 \cdot\!> l_1$

In this manner we obtained the six relations over the set A;
it can be seen which operations are parallel and which are serial to each other. Note that we could have equally said that, since neither $l_2 \cdot\!> l_1$ nor $l_1 \cdot\!> l_2$, we have

$$l_1 \doteq l_2$$

In this case we would have had the following six relations instead:

a') $l_4 \cdot\!> l_2$

b') $l_4 \cdot\!> l_3$

c') $l_3 \cdot\!> l_1$

d') $l_1 \doteq l_2$

e') $l_4 \cdot\!> l_1$

f') $l_3 \cdot\!> l_2$

The difference is between (d') and (d), and, (f') and (f); the rest are the same. The important thing is that no matter which set of relations we choose, the behaviour of the network is maintained; this is because of the definition (1) which corresponds to the behaviour of trees, and also because the axiomatic properties given at the beginning of this section truly represent parallel and sequential operations and the inter-relation between them.

11. Ordering the Operations

11.1 Open Trees

As our natural constraint is that: using a single processor, only one operation can be carried out at a (Computation) time, we should like to arrange the operations in a sequence such that the output signals obtained, as a result of performing them in that order, would correspond to the behaviour of the tree.

An operation x CAN be carried out when all operations $y \in A$ have been performed,

where $A = \{y \mid x \cdot\!> y\}$

The other constraint is that x SHOULD be carried out before all operations, $Z \in B$, have been performed, where $B = \{z \mid z \cdot\!> x\}$

This suggests a Computation Time interval, understood by $z \cdot\!> u \cdot\!> y$,

where $u \in C$

and $C = \{u \mid u \doteq x\}$

during which x could be performed. We note that all operations $u \in C$ are parallel to x.

The important result of the above analysis is the following assertion: THE BEHAVIOUR OF AN OPEN TREE CAN BE SIMULATED USING A SINGLE PROCESSOR. This assertion has two cornerstones; the first is axiom II, which as mentioned earlier, establishes one of the two relations, $\cdot\!>$ or $\doteq$, between any two operations belonging to a set of operations corresponding to a tree. The second is the fact that, using a single processor, a primitive element can be simulated.

11.2 Closed Trees

As mentioned in Section 7.4, the functional form of a closed tree is repetitive, i.e. once the output signals are obtained, the ones on thc closing terminals have to be fed back, and the process repeated. The best possible correspondence between a digital system and its model is obtained if the behaviour of the Simulator is such that the state of the model is re-evaluated at every point in Simulation Time. This repetitive evaluation causes the signals on the closing terminals to be fed back to the network. In this manner the behaviour of a closed tree is simulated using a single processor. Clearly, the Simulation Time is advanced at the end of each cycle.

By now the role of the linear description, in connection with the implementation of the Description System, has become apparent. Since in a linear description, the root of the tree is described after the sub-trees of that tree are described, the execution of the statements from top to bottom would simulate the behaviour of the open trees; the Simulator repeats its control cycle, and thus the behaviour of the closed trees is simulated.

The user need not necessarily specify his description in a linear form since, using a pre-processor, any symbolic description of a network can be transformed into a linear description.

12. Terminal Registers

The result of an operation may be needed at a later point in Computation Time. If so, we assume that there are imaginary 'terminal registers' connected to the terminals of a primitive element. In this form, c = and (a,b) could be thought of as:

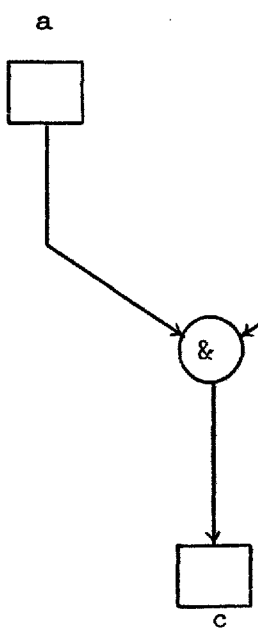


Figure 15. An and gate with its terminal registers

The name of a terminal register is the same as that of the signal it corresponds to. In this respect, one can give another definition for connectivity: two terminals are connected provided they share the same terminal register. The user can give a hint to the Simulator whether the result of an operation is going to be needed later. For example, from

d = <u>and</u> (<u>or</u> (a, b), <u>not</u> c);

it is understood that the intermediate signals, <u>or</u> (a, b) and <u>not</u> c, are not needed later.

In this manner, the operation of a primitive element can be looked at as a transfer of signals from input registers to the output register. This interpretation of the working of primitive elements take us to Chapter III, which discusses the generalised form of register transfer operations.

CHAPTER III

The Register Transfer Level

1. Introduction

At the Register Transfer Level (RT-Level), a digital system is imagined to be composed of registers connected through blocks of logic elements. Each block performs an operation on the contents of one or more input registers, and places the result in output registers, which may include some of the inputs.

At the end of the previous chapter, we noted that the structure and working of a network, at the Level of Logic Elements, could be described in terms of terminal registers and register transfer operations (RT-operations). In this chapter we generalize such operations to include vectors, as well as scalars. As a result of this generalization, the operators of the Language are defined, which cover the sub-classes of primitive elements mentioned in the previous chapter.

Detailed attention is paid to Language features for describing a digital system at the RT-Level. Some of the topics considered in the previous section, such as types and constants, are reviewed here with more emphasis on linguistic features, and occasionally on implementation requirements.

2. Representation of Boolean Constants

In digital systems information is represented using entities which take one of the two possible states. When concerned with switches and lights, these are called on and off; in connection with logical operations, they are named true and false; when a numerical interpretation is meant, these are referred to as 1 and 0.

One or more of these constant sets should be selected in order to assign states to lights, switches, and other binary objects. Whichever is selected, the Description System can choose only one of these representations to output the states of objects, unless the System has already been told about the type of object concerned with the information to be output. This would require introducing declarations such as:

```
switch     s;
memory     m;
register   r;
```

together with keeping distinct descriptors for each type. The approach might obscure from the user the fact that all objects, s, m, and r are essentially of the same type, namely Boolean.

Another extreme solution would be to allow only one type, for example Boolean, and ask the user to interpret all his binary entities in terms of this type. Besides putting the burden of the interpretation task on the user, we have also decreased the clarity of descriptions. In order to overcome this, the user would be forced to insert comments. For example,

```
Boolean s;  --  s is a switch;
Boolean m;  --  m is a memory
Boolean r;  --  r is a register;
```

A compromise solution, adopted here, is to permit the user to make use of all types (Boolean, switch, memory, etc.) interchangeably. The System will then interpret them all in the same manner. The price paid for this is that the output information would be represented in only one form. Corresponding to this, all sets of binary constants {on, off}, {1, 0}, {true, false} may be used interchangeably. We select the binary constants 1 and 0 for outputting binary information.

3. The unknown value

The question is whether to define the logical operators over two-valued or three-valued space. At a low level, a digital system can be thought of as a number of bistables connected through logic elements. When the system is switched on, the bistables take on one of the two possible states at random. Assuming that there are no faulty elements, or disconnections, and that other physical requirements such as the fan-out of the elements are satisfied, the system works on a two-valued space. Thus the only advantage in the inclusion of an unknown value would be the simulation of faults and other undesired effects due to non-initialization of storage elements which should have been initialized. Fault simulation may be done by defining faulty elements, with the aid of the element definition facility. In order to simulate the effect of non-initialization, one may either assume that the output of an element is unknown unless all its inputs are defined, or adopt some other arrangement which propagates the unknown value under certain conditions, so that the user becomes aware of the effects. However, this approach has shortcomings when dealing with networks whose working is independent of the initial values of their storage elements. If all storage elements were initialized to an undefined value by the System, then under this arrangement, the working of the model would not correspond to that of the digital system. Therefore we consider the logic elements to operate on a two-valued space.

4. Arrays

To identify one of many distinct objects, events, or operations, an array of binary entities must be used. In its simplest form, an array is a linear list, also called a vector. A binary vector may be interpreted numerically, or as a string of bits, an instruction, an address, or otherwise.

Main memories may be regarded as two dimensional arrays, also called matrices. As such, they are identified by the number of words they contain, and the number of bits per word. In the simplest case, the main memory has one address register which holds the address of the word to be written to or read from the memory, and one buffer register which holds the corresponding item of data. The memory read/write time is the same for all words. This type of memory may be declared as a <u>Boolean</u> <u>matrix</u>. More complex and special purpose memories such as stack, associative, multi-access, modular, etc. should be described explicitly.

4.1 The Type Integer

Integer entities are at a higher level than the binary ones - an integer can be represented by a binary vector. We therefore introduce the type <u>integer</u> so that arithmetic operations can easily be described at a high level. In input and output, integers are represented in the conventional decimal form. Arrays of integers do not seem to be required and we therefore exclude them.

4.2 Array Declaration

Arrays are declared by naming their type, and the lower and upper bounds for each dimension. Arrays and scalars may be declared together, for example

```
    switch  console(0:15), start, stop, power;
```

It is often required to refer to a sub-array of an array throughout a description. For example, an instruction in a computer may be composed of the following fields: operation-code, format, and address part, as defined below:

```
    register instruction(op-code(0:4), format, address (0:9));
```

Some of these fields may in turn contain sub-fields; for example, the address field may be regarded as composed of an indirect addressing flag, an index register indicator, and the static address part.

```
    register instruction(op-code(0:4), format, address(ia,ix(1:2),
                                  s-address(1:7)));
```

Sub-registers, format and ia, are of one bit length only. The above method for declaring arrays is called the nested-top-down method. Sometimes it is convenient to declare sub-arrays by giving the boundary bit positions that they occupy in the array, rather than by declaring the array in terms of its constituent sub-arrays. This is called the simple-top-down method, e.g.

```
    register double-word(0:31);
    sub-array word(0:15) = double-word(0:15),
              next-word(0:15) = double-word(16:31);
    sub-array instruction(0:15) = word(0:15);
```

Up to now the register, word, has had only one interpretation, i.e. an instruction. One can easily introduce other interpretations, e.g.

```
    sub-array fixed-point(0:15) = word(0:15);
    sub-array sign = fixed-point(0),
              magnitude = fixed-point(1:15);
```

and similarly for the register double-word:

```
    sub-array floating-point(0:31) = double-word(0:31);
    sub-array signs = floating-point(0),
              mantissa(0:23) = floating-point(1:24),
              exponent(1:7) = floating-point(25:31);
```

A picture of the structure which has been constructed is given in Figure 1. For clarity the picture is presented in three parts.

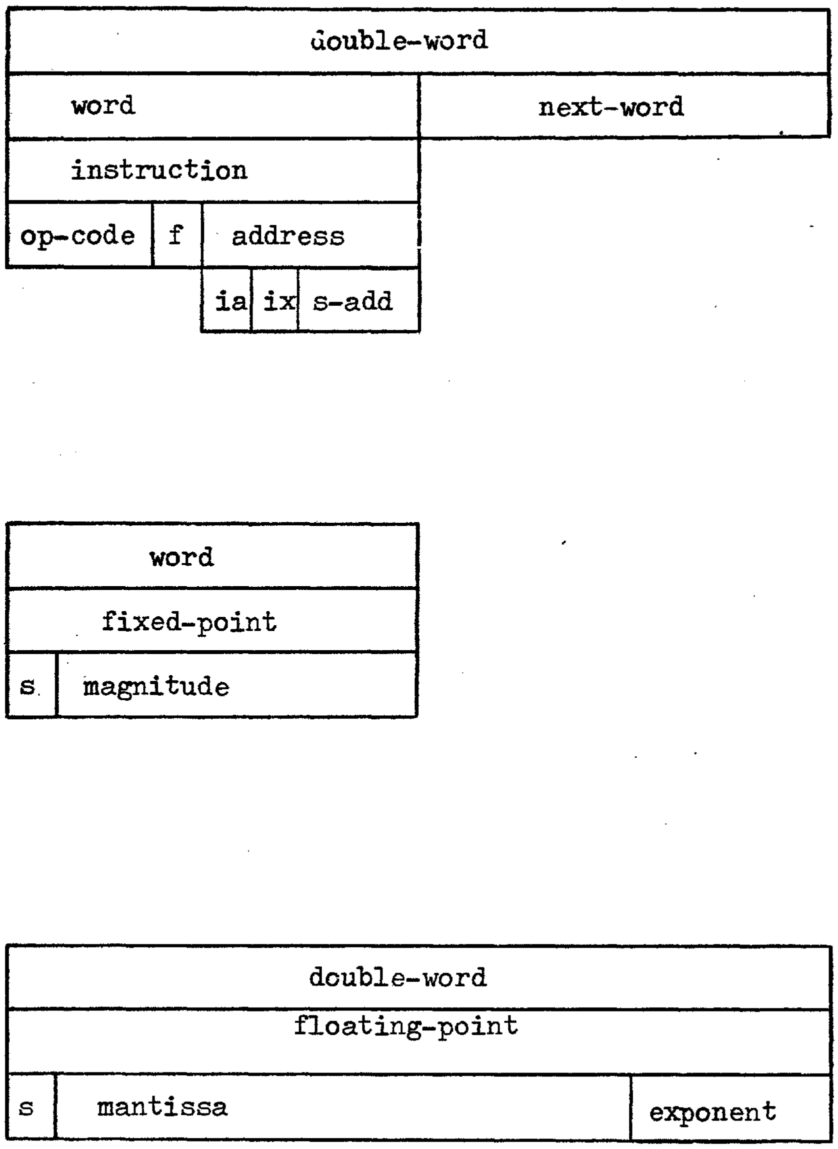


Figure 1. Interpretations of a double-word

Such constructions are ambiguous in the sense that referencing the name of an array, in general, does not convey the full meaning of the contents of that array. For example, word, may have different meanings such as instruction or fixed-point. This ambiguity is due to the very fact that a string of bits in a memory location could be interpreted in different ways: it could be an instruction, a data item, the second word of a double-word instruction, etc. Only the context of reference can clarify the meaning.

4.3 Array referencing

4.3.1 Referencing Sub-arrays which are Declared

There are two choices:

a) The sub-array might be referenced through the hierarchical structure it was declared in, e.g.

```
ix(address(instruction))
```

b) The sub-array might be called by its name only, e.g.

```
ix
```

In the first method two sub-arrays not belonging to the same array may have the same name, i.e. a certain name may give different meanings depending on the context, e.g.

```
a(c),
a(b).  (a, b, c are arrays)
```

This is a useful advantage when one has to declare many names, and wishes to use similar ones. However, as our declarations are not numerous, and because of its simplicity, we adopt the second method.

4.3.2 Referencing Sub-arrays which are not Declared

If references to a sub-array do not occur frequently, one may choose not to give it a symbolic name. The same notation used earlier can serve to construct sub-arrays or cascaded arrays, e.g.

```
instruction(0:4)
instruction(6:15)
op-code:address
word:next-word
```

In the last two examples, the colon sign serves to concatenate arrays. Dynamic references fall into this category, e.g.

```
word(n:m)
op-code(i):instruction(j:P):ix
```

4.4 Matrices

A matrix is an ordered list of vectors of the same dimension. In digital systems main memories and groups of registers are examples of matrices. When dealing with a register, one part of it cannot in general play the role of another part, for they usually perform distinct duties. By contrast, in a list of registers forming a matrix, one member may be replaced by another one. The only reason for having several of them instead of one is to increase the speed of the system. Although there is no exception to this general rule in the case of memories, i.e. all memory words perform the same function, there are exceptions with regard to registers. For example, in some computers certain index registers can also be used as accumulators.

Due to this interchangeability of registers and memory words among each other, there is no need to introduce elaborate facilities, as in the case of vectors, for constructing two dimensional structures. The fact is that one member of a list of registers is not going to be interpreted in a different way than another member, except in some special cases.

4.4.1 Declaration and Referencing of Matrices

We extend the notation introduced earlier for one-dimensional arrays in order to reference vectors forming a matrix. Also the special cases discussed in the previous section are catered for. Like vectors, matrices are declared by specifying the bounds for each dimension, e.g.

```
memory matrix M(0:8191, 0:15);
register matrix index-reg(1:3, 0:15);
```

In order to refer to the kth element of a list of vectors, the integer k is written in place of the first index, and the second index is ommitted, e.g.

```
m(i,)    ith memory word
index-reg(2,)    the second index register
```

Parts of such vectors may also be referenced in the same manner as before, e.g.

```
index-reg(2,0:4)   the first five bits of the second index register.
```

A vector belonging to a matrix may be given a symbolic name throughout a description, e.g.

```
sub-array  accumulator(0:15) = index-reg(3,);
```

5. Constants

5.1 Declaration

Integer constants may be given symbolic names, e.g.

<u>constant</u> add = 1, sub = 2, mult = 3, div = 4;

5.2 Specification

There are two types of constants, Boolean and integer; the latter is represented in the conventional decimal form. For the sake of simplicity, we allow Boolean constants to be specified in binary, octal, or hexadecimal form, as well as other forms discussed in Section 2, e.g.

952	a decimal constant
<u>bin</u>(0,0,1)	a binary constant
<u>oct</u>(7,7,7)	an octal constant
<u>hexa</u>(f,f,f,f)	a hexadecimal constant

6. Array Operations

As stated earlier in section 4.4, we need only be concerned with operations on one dimensional arrays. The operators and their characteristics are described in Table 1. The type of operand indicated here is that which the operator expects. If the operand is not of the required type it will be transformed according to the following rules: an integer is converted to an unsigned binary number represented by a Boolean vector; conversely, a Boolean vector is treated as an unsigned binary number and transformed into an integer.

If the user wishes to regard the numbers as signed, he may do so by placing a directive at the beginning of description, which indicates whether negative numbers are in two's or one's complement. The operator <u>int</u> in general takes a Boolean vector as its operand and transforms it into a signed integer. Similarly, the operator <u>bln</u> may be employed to convert a signed integer into a Boolean vector.

If the result of a conversion is a Boolean vector, its dimension is determined in accordance with the context. It should not be overlooked that we permit implicit type conversion at the cost of not forcing the user to distinguish between the state of a register and its interpretation.

In arrays, the element with the lowest index is called the first element. For positional and directional referencing this element is assumed to be the leftmost one, and in connection with binary numbers it is the most significant digit.

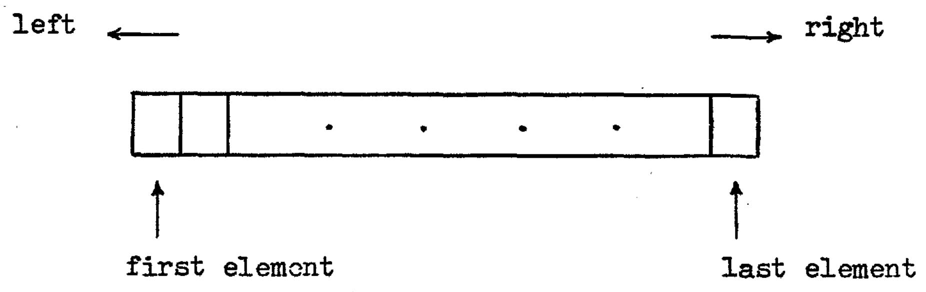


Figure 2.

Let $s_1, s_2, \ldots\ldots, s_m, x$ be scalars,

and $v_1, v_2, \ldots\ldots, v_m, y$ be vectors of length n,

and <u>asop</u> be an associative operator,

and <u>ident</u> (<u>asop</u>) be the identity operand for the operator <u>asop</u>.

Associative operators are allowed to have multiple operands, as shown below for scalars:

$x = \underline{asop}(s_1, s_2, \ldots\ldots, s_m)$ is the same as

$x = \underline{asop}(s_1, s_2, \ldots \underline{asop}(s_{m-1}, s_m))$

operator				number of operands			type of operand and [result]		associa-tive
symbolic	graphic	category	operation	monadic	diadic	multiple	Boolean	integer	
plus	+	arithmetic	addition	✓	✓	✓		✓[✓]	✓
minus	-	-	subtraction	✓	✓			✓[✓]	
mult	*	-	multiplication		✓	✓		✓[✓]	✓
div	/	-	division		✓			✓[✓]	
rem		-	remainder		✓			✓[✓]	
and	&	logical	and	✓	✓	✓	✓[✓]		✓
or	v	-	or	✓	✓	✓	✓[✓]		✓
not	~	-	not	✓			✓[✓]		
nand		-	nand	✓	✓	✓	✓[✓]		
nor		-	nor	✓	✓	✓	✓[✓]		
eqv	≡	-	equivalence	✓	✓	✓	✓[✓]		✓
eor		-	exclusive or	✓	✓	✓	✓[✓]		✓

Table 1.

operator				number of operands			type of operand and [result]		associa-tive
symbolic	graphic	category	operation	monadic	diadic	multiple	Boolean	integer	
lt	<	relational	less than		√		[√]	√	
le	≤	-	less than or equal to		√		[√]	√	
eq	=	-	equal		√		[√]	√	
ne	≠	-	not equal		√		[√]	√	
ge	≥	-	greater than or equal to		√		[√]	√	
gt	>	-	greater than		√		[√]	√	
rshift		shift	shift right	√	√		for shift operaters, the first operand is Boolean, and the second operand, if present, is integer. Type of result is Boolean		
lshift		-	shift left	√	√				
rcirc		-	circulate right	√	√				
lcirc		-	circulate left	√	√				
bln		special	convert to Boolean	√			[√]	√	
int		-	convert to integer	√			√	[√]	

Table 1 - (continued)

and for vectors

```
    y = asop(v_1,v_2,......, v_m) is the same as
        for i = 1 to n do
            y(i) = asop (v_1(i), v_2(i),......, v_m(i));
```

(see Section IV.8.1 for semantics of for structure.)

If an associative operator is given a vector as its only operand, the operator is distributed through the elements of the vector as shown below:

```
    x = asop(v_1) is the same as
    x = ident(asop);
    for i = 1 to n do
        x = asop(x, v_1(i));
```

In addition to associative operators, nand and nor are also allowed to have multiple operands; their operation in this rôle is described below:

nand l	is the same as	not and l
nor l	is the same as	not or l

where l is a list of operands, possibly only one vector.

The shift operators take a vector as their first operand and an integer as their second operand. The operation is then performed as many times as specified by the second operand. If the operation is to be done only once, the second operand may be omitted. Examples are:

<u>rcirc</u>(v_1, 2)   circulate v_1 two positions to the right.

<u>lshift</u> v_1 shift v_1 one position to the left.

The only case in which the operands of an operator can have different dimensions is when one operand is a scalar. The scalar is then distributed through all elements of the vector as shown below:

```
y = op(x, v1) is the same as
    for i = 1 to n do
        y(i) = op(x, v(i));
```

where <u>op</u> is an operator.

7. Expressions

<expression> ::= <simple expression>|<Boolean expression>|
                 <integer expression>|<delayed Boolean expression>|
                 <element designator>

<simple expression> ::= <constant>|<variable>

<Boolean expression> ::= <Boolean operator> <operand list>

<operand list> ::= <source vector>|(<source vector>{,<source vector>}*)

The simplest form of an expression is a constant or variable in which case a connectivity is established, or a reference to the contents of a variable, as defined in Chapter II. For a detailed syntactic definition of variables and constant refer to Appendix C.

An expression defines the structure and behaviour of a network. Among the structural properties the network inputs are given; the outputs are however not named. This is a feature of expressions which permits the specification of connectivity without naming the signals at the interconnections. An expression, in general, returns a vector as its result, e.g.

1) <u>register</u> a,b;
   <u>and</u>(a, <u>not</u> b)

2) <u>register</u> a(1:16), b(1:16);
   <u>and</u>(a, <u>not</u> b)

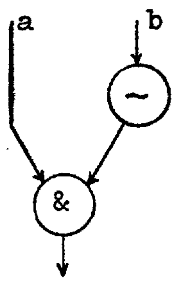


Figure 3. Graphic description

In both examples a and b are inputs. The first example returns a vector of length 1 as its result, while the second returns a vector of length 16. The signal in between the two gates is not named.

As was shown earlier in Chapter II, if a good correspondence between graphic and symbolic description of trees is required, it is better to represent the expressions in prefix notation; this also serves for describing elements with multiple inputs, e.g.

<u>nand</u> (a,b,c)

However, the prefix notation is not suitable for simplification, and addressing elements of arrays. The problem could partly be solved by using the prefix notation in general, and employing the infix notation for indexing; however, this would not be a good solution because of the non-uniformity introduced in the structure of expressions.

Boolean, shift, and relational expressions return Boolean results. The first two take Boolean operands, whereas the last one takes integer operands. The results and operands of arithmetic expressions are both integer. Other types of expressions will be discussed in Chapter V.

8. <u>RT-operations</u>

<source vector> ::= <exp>{:<exp>}*

<destination vector> ::= <variable>{:variable>}*

<RT-operation> ::= <destination vector> = <source vector>

The expressions outputs may be concatenated to form a source vector; Similarly variables may be concatenated to form a destination vector. Neither of these vectors is named. A 'register transfer operation' specifies the outputs of a network. It may also be looked upon as a mapping from a set of input vectors into a set of output vectors whose concatenation is called the destination vector.

Elements concatenated by the colon sign should be of the same type. If the number of elements of the source vector is equal to that of the destination vector, a one-to-one correspondence can easily be established; if the source vector has more elements than the destination vector, the extra elements at the left hand side are ignored; if the source vector has fewer elements, it is expanded to the left by repeating its leftmost element.

Some examples of RT-operations are given below:

1) Boolean j,k,q,nq;
   q,nq = jkbs (j, k, clock);

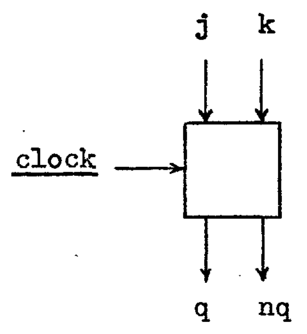


Figure 4. A JK-bistable

jkbs (JK-bistable) is a System-defined element; its outputs are q and nq (~ q). jkbs returns a vector of length 2; the first element is assigned to q, and the second to nq. When JK-bistables are cascaded (Fig. 8, or Fig. 10), the two inputs are not independent (k = ~ j).

In this case we use an abbreviated form as shown below:

q = <u>ajkbs</u>(j, <u>clock</u>);

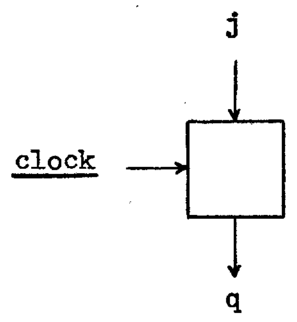


Figure 5. Graphic description of <u>ajkbs</u>

2) <u>register</u> a, b;

a:b = b:a;

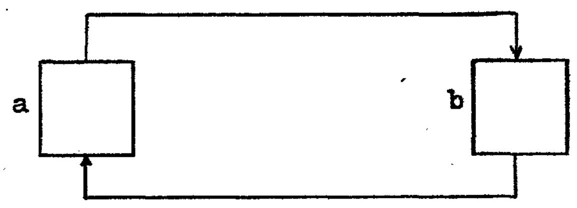


Figure 6. The contents of a and b are swopped upon the receipt of the controlling signal

3) <u>register</u> x(1:16), y(1:16), z(1:16);

x:y:z = y:z:x;

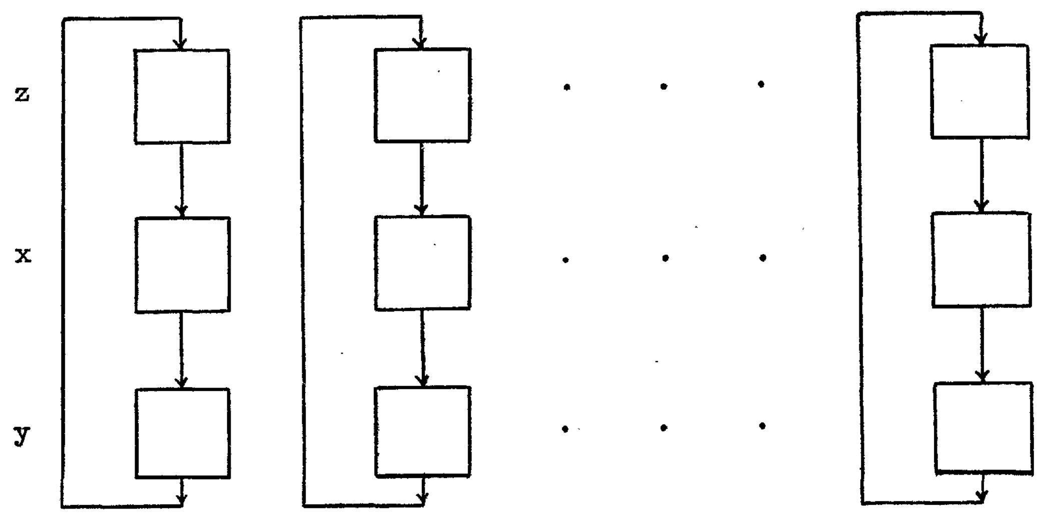


Figure 7. Upon activation, the contents of x, y, and z are rotated.

4) register a,b,c,d;

```
b:c:d:a = a:b:c:bin 0;
```

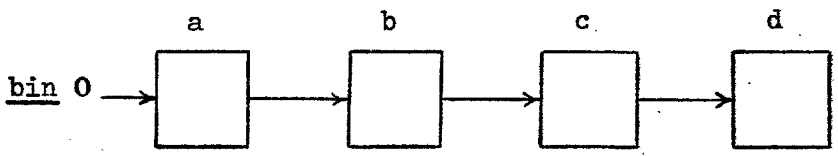


Figure 8. A 4 stage logical shift register

5) register a,b,c,d;

```
b:c:d:a = a:b:c:a;
```

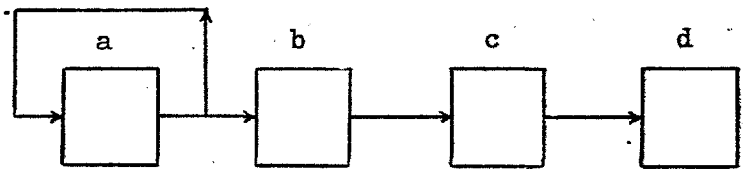


Figure 9. A 4 stage arithmetic shift register

6) register a(10);

```
a = rshift a(1:7):~a(8):a(9:10);
```

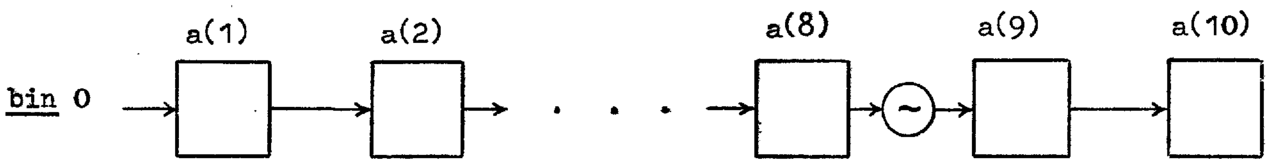


Figure 10. All positions of register, a, are shifted one place to the right, with the exception of the 9th position which receives the negation of the 8th position.

7) <u>switch</u> x(1:4);

   x = <u>off</u>, <u>off</u>, <u>on</u>, <u>off</u>;

   switch x is initialised.

8) <u>register</u> x(1:16), y(1:16),a;

   y = <u>and</u>(a,x);

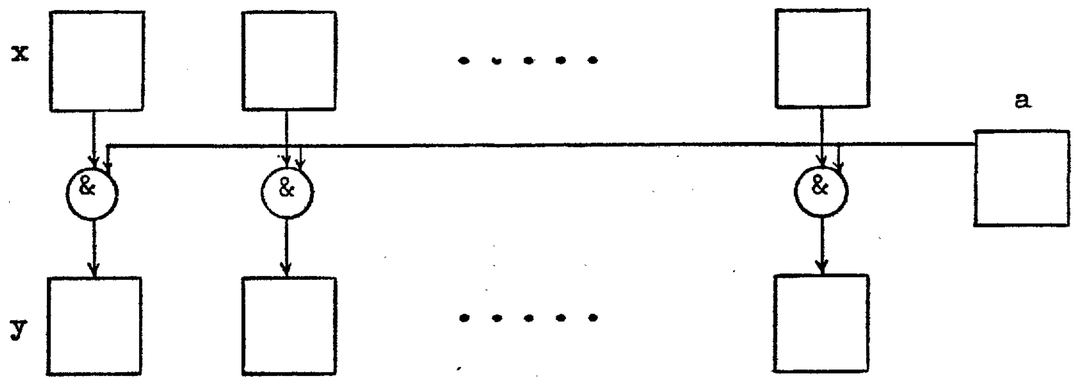


Figure 11. The control signal, a, gates a transfer from register x to y.

CHAPTER IV

The Control Level

1. Introduction

Up to now we have dealt with the operations without being concerned about when an operation should - or does - take place. At this level, control structures are introduced which can be used to impose an ordering on operations - an operation may be invoked at a particular step in a sequence of steps, or at a certain time, or in general when certain conditions are satisfied. Such conditions may well be a function of the previous state of the system.

The chapter starts with the introduction of the concept of a controlled RT-operation; two basic ways of ordering the operations, namely parallel and sequential are then discussed, and attention is paid to systems expressed in such terms. The latter part of the chapter is concerned with higher level control structures, such as conditionals and loops, which are frequently used for writing procedural algorithms.

Furthermore, the hardware implementation of such structures is discussed in detail; thus once the user has described his ideas in these terms, he has general paths in sight by which to progress toward the implementation level, and features of his design are realized as progress continues. The choice of control structures is important since these are the tools in terms of which the user will tend to think.

At the levels near to the implementation, we are concerned with a subset of RT-operations involving only Boolean variables. In general, a 'controlled RT-operation' is denoted in the following way:

<control condition> → <RT-operation>

where

<control condition> ::= <Boolean expression>

A controlled operation is said to be 'activated' or 'invoked' when the 'control condition' for that operation becomes <u>true</u>.

2. Parallel Networks

<parallel network> ::= <RT-operation>|[{<RT-operation>}$^+$]

A 'parallel network' is enclosed in square brackets unless it is composed of only one operation. The operations will later be given the same control condition. By parallelism it is meant that the transfers take place in two steps:

1) all expressions (source vectors) are evaluated
2) destination vectors take on their values simultaneously.

That is, the textual ordering of the RT-operations in a parallel network is immaterial. Examples of parallel networks are given below:

<u>register</u> a(1:4), b(1:4), c(1:4), d;

1) a = b;
2) [a = b; b = a; d = <u>and</u> c;]

A parallel network is said to be activated when its control condition becomes <u>true</u>, in which case all the RT-operations composing the network are activated. For example, one of the results of activating (2) is that the contents of registers a and b are swapped.

3. Sequential Networks

<sequential network> ::= <parallel network>|({<parallel network>}$^+$)

A 'sequential network', which is an ordered list of parallel networks, is enclosed in round brackets unless it is composed of only one parallel network. Examples are:

```
register  a(1:4), b(1:4), c(1:4), d;

1)   a = b;
2)   [a = b; b = a;]
3)   (d = not b;[c = d; a = b;] d = or a;)
```

Since a sequential network is an ordered list, we can talk about the 'successor' or 'predecessor' of a member of the list. When the operation of each member of the list terminates it invokes its successor. The process continues until the last member of the list is activated. The first member is invoked when the network control condition becomes true.

4. Controlled Networks

<controlled network> ::= <control condition> <control sign>
<sequential network>|<sequential network>

<control sign> ::= →

A 'controlled network' is a sequential network with a control condition. Whenever the condition becomes true the network is activated. In the special case that a network has to be activated at all successive points in time, i.e. the control condition is always true, we may omit the control sign and condition. Examples are:

```
     Boolean w, x, y;
     register a(1:4), b(1:4), c(1:4), d;

1)   w → a = b;
2)   and(y, x) → [a = b; b = c;]
3)   and(y, or(w, x)) → (d = and c; [a = b; b = d;] d = or c;)
4)   true → d = not d;
5)   a = not b;
```

At this point we have to clarify what exactly is meant by 'whenever a condition becomes true'. Digitizing the time implies that in the most accurate case a control condition may be tested only at consecutive discrete points in time. This should not, however, be interpreted as a restriction, since the distance between two neighbouring points in time, hereafter called Δt, may be assumed to be so short that no event is missed. As illustrated below, if t_{m+1} is the successor of t_m the event occuring at x will be skipped.

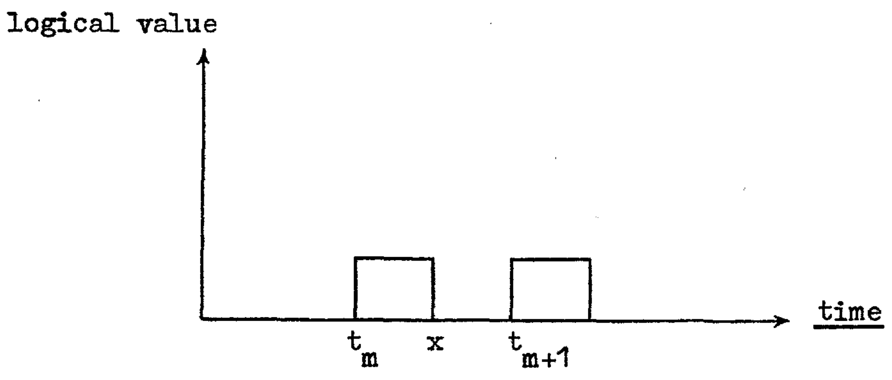


Figure 1. A sequence of events

In this respect, we state the following assumption: CONTROL CONDITIONS ARE TESTED AT ALL SUCCESSIVE POINTS IN TIME. Our Second Assumption at this level is that: ALL OPERATIONS IN A CONTROLLED SEQUENTIAL NETWORK, INVOKED BY ITS CONTROL CONDITION AT A CERTAIN TIME, WILL COME TO CONCLUSION BY THE NEXT POINT IN TIME.

The above assumptions are understood to be part of the semantics of a controlled network. They relieve the user from some timing problems so that he can concentrate on other aspects of his design. Nevertheless, at a lower level the designer must comply with certain requirements in order that the assumptions at the present level be maintained. Later these requirements will be dealt with.

5. General Networks

```
<general network> ::= {<controlled network>}+
```

A 'general network' is composed of several controlled networks, each of which behaves as described in the previous section. At every time point, firstly all control conditions are tested, and then the networks corresponding to the true control conditions are activated. An example of a general network is given below:

```
register a(1:4), b(1:4), c(1:4);
Boolean w, x, y;
    w → [c = add(a,b); a = sub(a,b);]
    x → (a = add(b,c); [b = c; c = a;])
    and (x,y) → x = false;
```

The two assumptions set out earlier apply to each member of a general network, in addition to our next assumption which is a consequence of the aggregation of several controlled networks. Before stating this assumption we need to establish the following definitions.

Two RT-operations are said to be 'directly joint' if they share only a destination register, and 'indirectly joint' if the output of one is an input to the other. If two operations are not directly or indirectly joint, they are said to be 'disjoint'. Two controlled networks are joint if they have two joint operations; otherwise they are disjoint. A general network is disjoint if all its members are disjoint. Let

$$w_1 \rightarrow A_1$$

$$w_2 \rightarrow A_2$$

be two disjoint members of a general network. If there exists a time point at which both w_1 and w_2 are true it is immaterial which network is activated - i.e. its operations take place - first. The conflict arises when the two networks are joint. Our third assumption is now stated to be: THE OPERATIONS OF JOINT NETWORKS TAKE PLACE ACCORDING TO THEIR TEXTUAL ORDERING, i.e. THE TOP ONE IS THE FIRST. This, of course, means that if A_1 and A_2 are directly joint through a vector s, the operation involving s in A_2 has a higher significance than that in A_1, since this is the one which is carried out last, and therefore its effect remains in the system for the present time interval.

Note that the Third Assumption gives rise to a new way of sequencing; for example the networks (1) and (2) below are the same:

1) Boolean w;

$$w \rightarrow (op_1\ op_2\ .\ .\ .\ op_n)$$

2) Boolean w;

$$w \rightarrow op_1$$

$$w \rightarrow op_2$$

.

.

.

$$w \rightarrow op_n$$

where op_i is an RT-operation.

In summary, a general network is an ordered list of controlled sequential networks with the assumption that if there exists a time point at which more than one sequential network is activated, the operations take place according to their textual ordering.

6. Systems

```
<system> ::= {<declaration>}*<system body>
<system body> ::= <initialization> end;|
                  start; <general network> end;|
                  <initialization> start; <general network> end;
<initialization> ::= <sequential network>
```

A 'system' is composed of a general network together with the necessary initializations required for the network to perform in the way it is expected to. Initializations conclude before the network begins to operate. As far as hardware implementation is concerned, they are done by manual switches. The following are two examples of systems:

```
1) Boolean clk;             -- a user defined clock;
         clk = false;       -- initialize the clock to false;

   start;
         clk = not clk;     -- at successive points clk changes state;
   end;

2) register a(1:4), b(1:16), c(1:16);
         a = bin(0,0,0,1);
```

```
start;
        -- shift left a and b three places;
        ~ a(1) → [a = lshift a; b = lshift b;]
        -- at the end transfer b to c;
        a(1) → c = b;
end;
```

7. Implementation of General Networks

In this section we discuss the problem of implementing a general network assuming that the implementation of a controlled RT-operation is already known. It should be pointed out that this is a fair assumption, since RT-modules are usually available to the user [8,21]. Nevertheless, if they are not available, it is easy to make them out of primitive logic elements such as JK-bistables, and, or, not, or similar gates. When we set out to implement a general network, the task will naturally result in the representation of the network at successively lower levels, where assumptions inherent in the high level representation turn into requirements that should be acknowledged and implemented in order to comply with those assumptions.

It is frequently desired to identify a certain step of an event composed of several steps. To this end, depending on the circumstances, some kind of 'sequencer' is used. The simplest one, called the synchronous sequencer is described below:

```
register s(1:M);
        s = bin (1,0,0, ....,0); -- initialize;
start;
        true → s = rshift s;     -- shift right at each time point;
end;
```

The above component may be used to distinguish between M steps of an event; when s(i) becomes true, the ith operation is identified. Furthermore, when all stages of s are false the end of the event is signalled. Equally, one can use an M + 1 stage sequencer where the last stage identifies the end of the event.

7.1 Reduction to Type α Networks

Let a general network be composed of controlled networks

$$w \rightarrow A$$

where w is the control condition and

$$A = (A_1\ A_2\ \ldots.\ A_M)$$

That is, A is a sequential network and each A_i is a parallel one. Using a sequencer we arrange that the M steps of A occur in order. This process demonstrates the property of sequential networks, that each one activates its successor. The new description is as follows:

```
register s(1:M);
        s = bin (1,0,0, .... 0);
start;
        true → s = rshift s;
        and (s(1),w) → A₁
        and (s(2),w) → A₂
                  .
                  .
                  .
        and (s(M),w) → A_M
end;
```

Let a network be composed of controlled networks

$$x \rightarrow B$$

where x is the control condition and

$$B = [B_1\ B_2 \dots B_N]$$

That is, B is a parallel network and each B_i is an RT-operation. This combination is called a 'type α network'. Thus, the first stage of the reduction to implementation level, has been reduction to a type α network.

We recall that our Second Assumption is that all operations initiated at a point in time will conclude by the next time point. One can see that it is easier to comply with this assumption in a type α network than in a general network because of the parallel-only nature of the operations. But we are not yet in a position to say that the Second Assumption has been complied with, since not all parallel networks A_i do take the same time to conclude. Let A_j be the one which takes the longest time. One way to solve the problem would be to adjust the First Assumption, i.e. to permit Δt to be at least as long as the time taken by A_j. This is not however a good solution, because it makes Δt dependent upon the behaviour of a certain network, and hence all other parts of the system run as slowly as this network does, which is contrary to the normal requirement from a design product that it should be as fast as circumstances allow.

Let $t(A_k)$ denote the time taken by A_k to conclude; this can be expressed as:

$$t(A_k) = n_k * \Delta t,\ n_k \geq 1$$

We now arrive at a new solution, that is to make the result of A_k available to its successor A_{k+1} after n_k time units as shown in the following segment. This is called the 'synchronous' method.

$$\vdots$$

$$\underline{\text{and}}(s(1),w) \rightarrow A_k$$
$$\underline{\text{and}}(s(+(1,n_k)),w) \rightarrow A_{k+1}$$

$$\vdots$$

where s(1) is the stage corresponding to A_k, and therefore $s(+(1,n_k))$ is the stage corresponding to A_{k+1}. This method, which is applicable in many cases, does not have the drawback of the first one, but it assumes that we already know how long A_k would take. When $t(A_k)$ depends on the data supplied to it, the following method, known as 'asynchronous', is used.

First we define an asynchronous sequencer or, more precisely, an asynchronous stage in a sequencer. Each stage of a synchronous sequencer is true for a time interval determined by an external signal, while for an asynchronous stage this length of time is determined by the component to which the stage corresponds. The asynchronous component is activated when its corresponding stage is set to true. When its operation ends, the stage is set to false; the succeeding stage then becomes true. Any number of synchronous or asynchronous stages can be cascaded together.

In summary, during the reduction to type α we saw that all the three assumptions remained invariant. This was because the form of representation did not change, i.e. the network was always represented as a list of controlled networks. As a result of this reduction, we showed how the requirements for complying with the Second Assumption could be implemented.

7.2 Implementation of Joint Networks

Let

$$\begin{array}{l} w_1 \rightarrow [\ \ldots\ op_1\ \ldots\ ] \\ w_2 \rightarrow [\ \ldots\ op_2\ \ldots\ ] \\ \cdot \\ \cdot \\ \cdot \\ w_N \rightarrow [\ \ldots\ op_N\ \ldots\ ] \end{array}$$

be a type α network where each op_i is an RT-operation, and due to the existence of certain joint operations, op_i's must take place according to their textual ordering. We integrate all the joint operations to obtain:

$$\underline{\text{or}}\ (w_1, w_2, \ldots, w_N) \rightarrow \begin{array}{l} (\underline{\text{if}}\ w_1\ \underline{\text{then}}\ op_1 \\ \ \underline{\text{if}}\ w_2\ \underline{\text{then}}\ op_2 \\ \ \cdot \\ \ \cdot \\ \ \cdot \\ \ \underline{\text{if}}\ w_N\ \underline{\text{then}}\ op_N) \end{array}$$

This is a controlled sequential network which can be implemented using a sequencer composed of N stages. Although the above solution is a general one, it may not always be desirable because of the slow nature of the sequential network. The designer must search for more suitable solutions which better suit his special case.

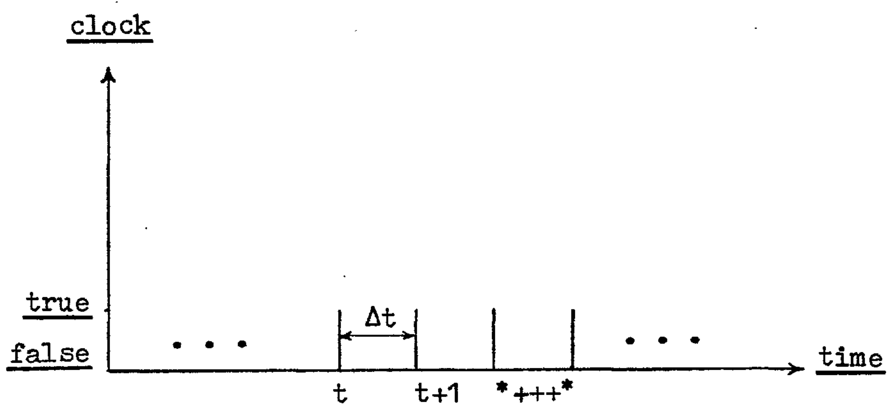
As will be seen later in the examples, the control conditions are such that very few of them may be <u>true</u> at the same time and there is little chance that the <u>true</u> ones have a shared register. A case which usually gives rise to the need for integration is when one of the control conditions is the constant <u>true</u>.

7.3 The clock

Our First Assumption was stated to be: control conditions are tested at consecutive discrete points in time. The result of this test is that controlled networks whose control conditions are true are activated at the forthcoming point in time. Therefore, the requirement can be re-formulated to become: the only assignments taking place at the forthcoming point in time are to the destination registers in controlled networks with true control conditions. If there exists a variable which is true at every successive time point, and false otherwise, the problem is solved. Let us call this variable clock; then, the condition for the assignment is:

and(clock, control condition)

This, of course, means that the storage elements employed as destination registers should offer the facility for conditional assignment. A clock cannot be described in terms of our primitives, i.e. controlled RT-operations. However, it is easy to make such a component out of primitives lower than those we have ever considered. Its output signal varies according to the time as shown below:



* assignments take place

+ expressions are evaluated

Figure 2. The clock output signal

There are variations in the type of storage elements used as registers, and in the waveform of the clock; these are discussed in Chapter V. In all cases the general idea is to cater for the implementation of the First Assumption.

One would like to speed up the computation by increasing the rate of the clock pulse. This, however, would shorten the time for evaluating expressions (Δt), which calls for the use of elements with a shorter inherent time delay - one of the factors to be considered when compromising between cost and speed.

8. Higher Level Control Structures

```
<sequential network> ::= <conditional structure>|
                         <unconditional structure>
<conditional structure> ::= <if structure>
<unconditional structure> ::= <parallel network>|<block>|
                              <for structure>|
                              <case structure>|
                              <loop structure>|
                              <miscellaneous structure>

<parallel network> ::= <RT-operation>|[{<RT-operation>}+]
<block> ::= ({<sequential network>}+)
```

Control structures are essentially at the same syntactic level as parallel networks. Therefore, we have extended the definition of a sequential network to include these. As might be expected from the syntactic form, all control structures need to be activated. They may be either explicitly conditional, like the if structure or unconditional like the rest. With this modified form of definition, a sequential network may be as simple as an RT-operation, or as complex as the structures suggested by a block.

Blocks are enclosed in round brackets which in this context are a sign of sequentiality. The <u>for</u> structure differs from all other control structures in that it is used only to define repetitive constructions.

8.1 Repetitive Constructions

```
<for structure> ::= <for clause> do <sequential network>
<for clause> ::= <for list>|<for list> step <index step>
<for list> ::= for <for index> = <initial value> to <final value>
<for index> ::= <integer variable>
<initial value> ::= <integer constant>
<final value> ::= <integer constant>
<index step> ::= <integer constant>
```

Repetitive constructions may be described with the aid of the <u>for</u> structure which defines zero or more instances of a sequential network. Each instance differs from the previous one in the value of the <u>for</u> index. The index starts at the initial value and is increased by the specified step until it reaches its final value. If no step is given, the index is increased by 1. Some examples of the <u>for</u> structure were given in Section III.6. Two more examples are given below:

```
1)  register   a(1:16),b,c(1:16);
          for i = 1 to 15 step 2 do [
              a(i) = b; c(i) = b;]
```

The equivalent graphic description is given in Figure 3. This network, when activated, sets all the odd positions of registers a and c to the value contained in b. Notice the correspondence between the parallel nature of the graphic and symbolic descriptions.

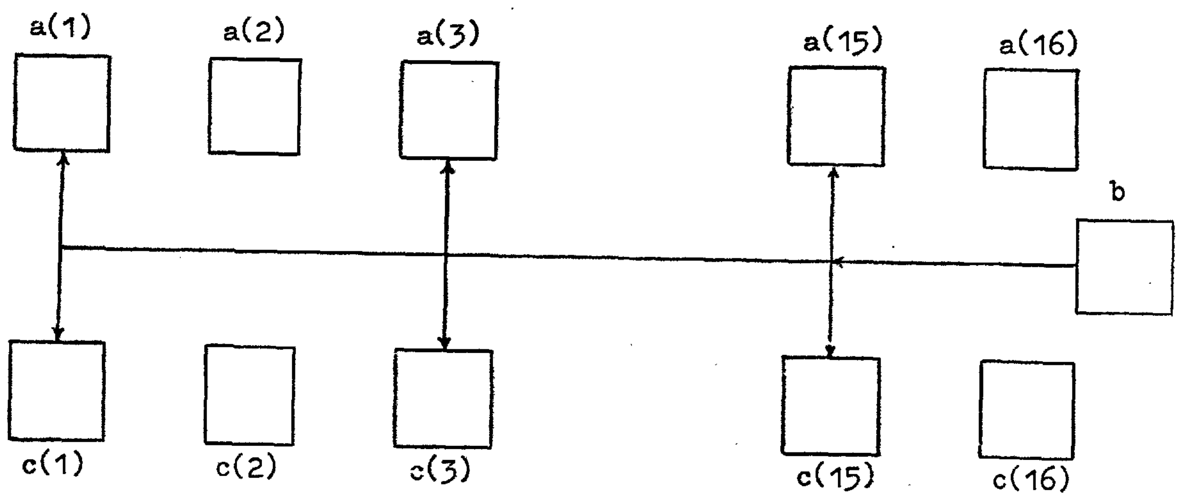


Figure 3.

```
2) Boolean  a(1:16), na(1:16);
     for i = 1 to 15 do
         a(+(i,1)): na(+(i,1)) = jkbs(a(i), na(i), clock);
     a(1): na(1) = jkbs(a(16), na(16), clock);
```

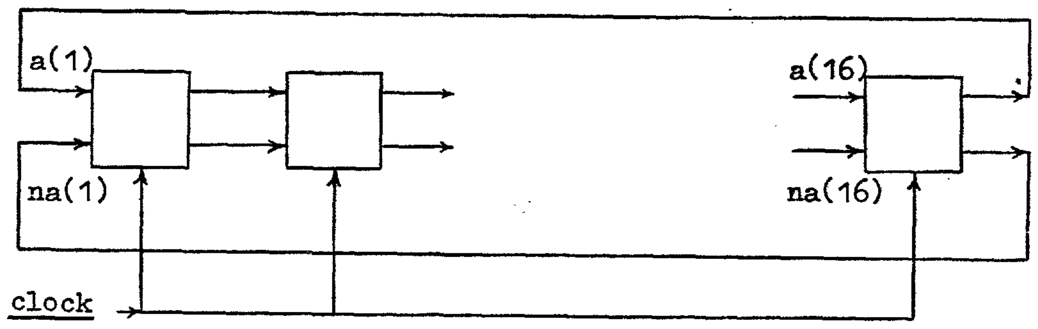


Figure 4. 16 JK-bistables connected in a circular way

8.2 Implementation of Control Structures

As will be seen below, control structures can be simply implemented in hardware; this is particularly interesting since experience in developing algorithms has shown how frequently these structures occur.

Control structures are essentially asynchronous, since the time they take to complete their operation varies from one activation to another. At the end of their operation a completion signal, called comp, is sent which allows the successor of the control structure to be activated. It should always be reset to false before the asynchronous device is invoked.

8.3 The if structure

```
<if structure> ::= <simple if>|<simple if> else <sequential network>
<simple if> ::= <if clause> <unconditional structure>
<if clause> ::= if <control condition> then
```

If the control condition is true the unconditional structure is activated; otherwise, the sequential network, if any, is invoked. Examples are:

```
        register  a(1:4), b(1:4), c(1:4),d;
        Boolean x, y;
    1)  y → if x then (a = b; b = c;) else [a = b; b = a;]
    2)  if x then d = and a;
```

8.4 Implementation of the if Structure

We augment both the unconditional structure and the sequential network by the operation comp = true; in parallel to their last operation. If no sequential network is given, the result of augmentation is the operation itself. The control condition for the augmented unconditional structure is given in the if clause, and its negation is the control condition for the newly formed sequential network. For example the augmented forms of (1) and (2) are:

```
    1)  y → if x then (a = b; [b = c; comp = true;])
                  else [a = b; b = a; comp = true;]
```

```
2)  if x then [d = and a; comp = true;]
         else comp = true;
```

When the if structure involves only RT-operations, like the two examples considered above, it is easier to implement it in a synchronous manner. For example, (1) could be implemented as:

```
and (y, x) → (a = b; b = c;)
and (y, not x) → [a = b; b = a;]
```

8.5 The case Structure

```
<case structure> ::= case <selector> of <case body>
<selector> ::= <integer variable>
<case body> ::= [ {<case primary>}+]
<case primary> ::= <label>{,<label>}*→ <sequential network>
<label> ::= <integer constant>| others
```

The sequential network whose label is equal to the value of the selector will be activated. The labels must be distinct. If no such network exists, the one labelled others will be activated provided this label is specified, e.g.

```
constant  add = 1, sub = 2, . . . , shift = 10; -- operations;
integer   op-code; -- operation code;
register  acc(1:16); -- accumulator;
register  afr(1:16), -- arithmetic factor register;
register  error; -- error indicator for invalid operation codes
          case  opcode of [
                add → acc = add(acc, afr);
                sub → acc = sub(acc, afr);
                .
                .
                .
                others → error = true;]
```

8.6 Implementation of the case Structure

First it must be decided how to represent the selector. Among various ways of doing this, we choose the base infinity (∞) representation, whereby to represent n distinct values, n distinct digits are needed. In the binary coded form of base ∞, the element s(i) of the Boolean vector s is set to true to represent the ith digit. This representation, which is extremely redundant, has however the advantage of being easily interpretable; in fact the information is already in the final interpreted form. Furthermore, the operations addition and subtraction can be easily performed; in the special case of increasing or decreasing by one, they are simply done by shifting to the right or left. (Due to these advantages the representation is used for lift position-indicators.) There are simple ways of mapping other forms of binary representation onto base ∞ and vice versa. At present we use this base becuase of its ease of interpretation.

We augment all the sequential networks of the case structure by the operation comp = true; in parallel to their last operation. If no network labelled others exists, one will be created whose only operation is comp = true;. Let Z_i be one such newly formed sequential network whose label is a constant and corresponds to s(i); then, s(i) is the control condition for Z_i. Let $s(k_1)$, $s(k_2)$, . . . , $s(k_n)$ be the elements which do not correspond to any of the constant labels; then,

$$\underline{or}(s(k_1),\ s(k_2),\ .\ .\ .\ ,\ s(k_n))$$

is the control condition for the sequential network labelled others.

Like the if structure, when the case structure involves only RT-operations it is easier to implement it in a synchronous manner.

8.7 Loop Structures

<loop structure> ::= <do structure>|<while-do structure>|<do-while structure>

There are three kinds of loops: do is a loop without a termination test; while-do has its termination test at the beginning of the loop; and do-while is used for a test at the end.

It is often needed to exit from a loop under certain conditions. A primitive element, named exit, is included for this purpose. As a result of activating this element, the successor of the loop is invoked.

In order to implement loops, a loop sequencer is used, as defined below for sequencing M steps of an event.

```
register  s(1:M);
    s = bin(1,0,0, . . . , 0);
start;
    true → s = rcirc s;
end;
```

The only difference between this sequencer and the one described in Section 7 is in the use of circulation instead of shifting. Loop sequencers may contain stages for asynchronous operations; this is needed, for example, when control structures arenested.

In all kinds of loops, two things must be done in order to exit: the loop sequencer must be stopped by resetting it to false, and the operation comp = true; activated. Since only one of the stages of the sequencer is true at a time, in order to reset the sequencer it is sufficient to reset the successor of this stage.

8.7.1 The do Structure

<do structure> ::= do <sequential network>

The network is repeatedly activated until an exit element is invoked, in which case the successor of the structure is activated. For example, as a result of activating the following sequential network all even locations of register f are set to false, in series.

```
register f(1:16);
integer i;
    i = 1;
    do(f = lcirc f;
       f = lshift f;
       if eq(i,8) then exit;
       i = plus(i,1);)
```

8.7.2 Implementation of the do Structure

In the implementation the exit element plays two roles: it resets the loop sequencer to false, and sets the register, comp, to true. These two operations can be done in parallel. A low level description of the previous example is given below. In this, s is a synchronous loop sequencer.

```
     register f(1:16), s(1:4), comp;
     integer i;
1-       [i = 1;
2-        s = bin(1,0,0,0);]
     start;
3-       true → s = rcirc s;
4-       s(1) → f = lcirc f;
5-       s(2) → f = lshift f;
```

```
6-      s(3) → if eq(i,8) then [s(4) = false; comp = true;]
7-      s(4) → i = plus(i,1);
    end;
```

There are a number of points to be noticed in this description. Firstly, the initializations are in parallel. Secondly, the exit element is translated into two parallel operations. Thirdly, the control conditions of lines 4 to 7 become true one at a time; this is typical of descriptions at this level. Finally, there is a point in time at which control conditions of both lines 3 and 6 are true, and also i = 8; then, two assignments to location s(4) of the register s occur. As mentioned earlier the one textually lower has a higher significance, which results in the resetting of the sequencer.

8.7.3 The while-do Structure

```
<while-do structure> ::= while <control condition> do
                                  <sequential network>
```

The network is repeatedly activated while the control condition is true. The condition is tested at the beginning of operations; therefore if it is false at the beginning the network is not activated at all.

8.7.4 Implementation of the while-do Structure

The while-do structure is a special case of the do structure where the exit test is performed at the beginning of operations. If the sequential network involves n steps, a loop sequencer composed of n + 1 stages is used. The first stage caters for testing the condition.

8.7.5 The do-while Structure

<do-while structure> ::= do <sequential network> while <control condition>

The network is repeatedly activated while the control condition is true. The control condition is tested at the end of operations; therefore the network is activated at least once.

8.7.6 Implementation of the do-while Structure

This is implemented as a special case of the do structure, where the exit test is performed at the end of operations. If the sequential network involves n steps, a loop sequencer composed of n + 1 stages is used. The last stage caters for testing the control condition.

9. An Example of Design

This section is concerned with an example of the design of a component with the aid of concepts discussed so far. First a high level description of the device to be designed is produced from its verbal description; this description is then successively reduced toward the implementation level.

We would like to design a component which repeatedly shifts to the left the contents of the register, acc, until the leftmost position contains a true bit. The integer i holds the count of shifts, and it is assumed that acc contains at least one true bit. The formal description is as follows:

```
register acc(1:16);
integer i;
    i = 0;
    while not acc(1) do [
           acc = lshift acc;
           i = plus(i,1);]
end;
```

The two operations of the while-do loop are stated to be in parallel since they do not interfere with each other. In order to make the exit condition more transparent, we now express the description in terms of a do loop.

```
register acc(1:16);
integer i;
    i = 0;
    do(if acc(1) then exit;
       [acc = lshift acc;
        i = plus(i,1);])
end;
```

At the next step we need a two-stage synchronous loop sequencer to implement the do loop. The first stage is concerned with the exit test and the second with the operations of the loop.

```
register acc(1:16), s(1:2), comp;
integer i;
    [i = 0;
     s = bin(1,0);
     comp = false;
start;
    true → s = rcirc s;
    s(1) → if acc(1) then [s(2) = false; comp = true;]
    s(2) → [acc = lshift acc; i = plus(i,1);]
end;
```

As mentioned in Section 8.7.2 the exit element is translated into two RT-operations: the first one resets the sequencer and the second sends the termination signal.

At the next level the decision must be taken how to represent i. We adopt the familiar base ∞ representation for the sake of simplicity in counting. Acc may be shifted to the left by at most 15 positions; therefore a total of 16 positions are required, the first one for representing 0.

```
     register acc(1:16), s(1:2), comp;
     register cnt(1:16); -- register to hold the number of shifts;
         [cnt = hexa(8,0,0,0); -- set count to decimal zero;
          s = bin(1,0);
          comp = false;]
     start;
1-       true → s = rcirc s;
2-       and(s(1), acc(1)) → [s(2) = false; comp = true;]
3-       s(2) → [acc = lshift acc; cnt = rshift cnt;]
     end;
```

This description is in the form of a type α network, where lines 1 and 2 are directly joint. We integrate the joint operations in order to obtain a disjoint type α network. Only the main body of the description is given below since the rest remains unchanged.

```
1-   true → s = rcirc(and(s(1), not acc(1)):s(2));
2-   and(s(1), acc(1)) → comp = true;
3-   s(2) → [acc = lshift acc; cnt = rshift cnt;]
```

The second line can be written in the following equivalent, but simpler form:

```
2-   true → comp = and(s(1), acc(1));
```

The description is now ready for final implementation. Remember that the control conditions for register assignments should be <u>and</u>'ed with the <u>clock</u>. The graphic description in terms of abbreviated JK-bistables is given below:

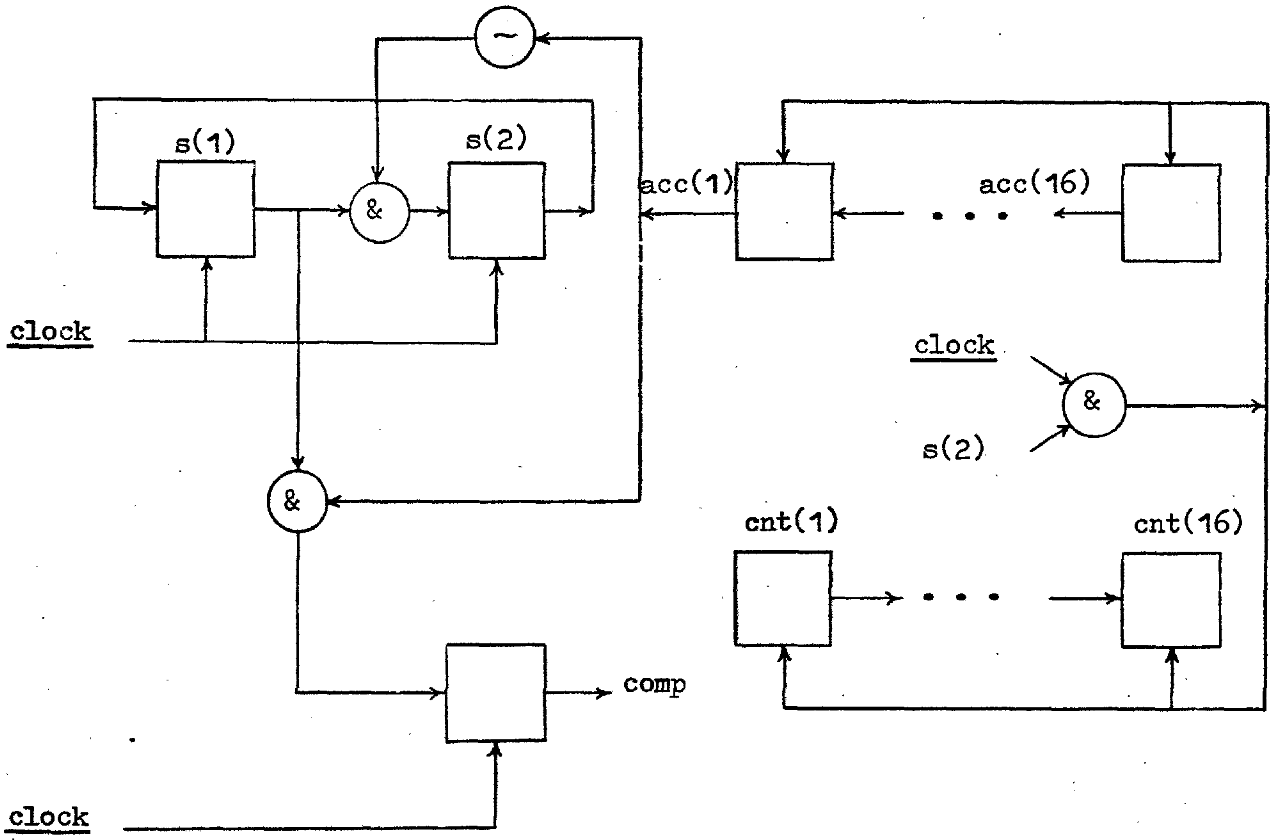


Figure 5. The component to shift left and count

The symbolic description is as follows

```
for i = 1 to 15 do
    acc(i) = ajkbs(acc(+(i,1)), and(clock,s(2)));
for i = 1 to 15 do
    cnt(+(i,1)) = ajkbs(cnt(1), and(clock, s(2)));
s(1) = ajkbs(s(2), clock);
s(2) = ajkbs(and(s(1), not acc(1)), clock);
comp = ajkbs(and(s(1), acc(1)), clock);
```

The network is activated when s(1) is set to true. The initializations should conclude prior to activation; also, the register acc should contain the bit pattern to be operated upon.

CHAPTER V

Miscellaneous Extensions

1. Introduction

Most of this chapter is devoted to the description of System- and user-defined elements. Analogies are pointed out between the rôles of such elements in a description language and sub-programs in a programming language; some examples are given to show how user-defined elements could be employed in extending the basic set of primitive elements available to the user, and in composition of networks.

The chapter concludes by stating that a description at any level could be looked at as if it had been given at the Control Level, and that the Control Level is thus the generalized form of all levels.

The System bistables are basically selected from the DEC Logic Handbook [18]. Their behaviour differs slightly from that of DEC bistables because of the digitization of time.

Syntactic details of the structures introduced in this chapter are given in Appendix C.

2. Logic Elements with Time Delay

It is quite common to consider a time delay of one unit for certain of the gates forming a network. Although the delay element, as demonstrated in Section II.9.1, can serve to introduce the desired time delay, a simpler way is suggested in this section, namely extending the set of operators of the Language (Table III.1) to include operators with one unit of time delay; the letter, d, in front of the name of standard logical operators means that the corresponding operator imposes the required delay. In this manner seven operators, viz. dand, dor, dnot, dnand, dnor, deqv, deor are formed which may be used along with the other operators of the Language to define networks or new elements. (Examples of these will be seen later in this chapter.) The expressions corresponding to these operators are called delayed Boolean expressions.

3. System-defined Elements

System-defined elements perform functions more complex than those of primitive elements. Usually it is better to leave the task of defining complex elements to the user so that they can be defined in accordance with the user's specific requirements. However, the inclusion of certain of such functions allows the System to implement them efficiently, as well as providing the user with additional facilities. In the following sections we shall describe two types of elements incorporated in the System - binary adding elements and bistables.

3.1 Binary Adding Elements

The four binary adding elements operate on, and result in binary numbers represented by Boolean vectors. They are:

1) add: this element takes two vectors of size n as its inputs, and returns their sum in a vector of size n + 1. The first element of the output vector holds the carry out, e.g.

```
register opr(1:16); -- operand register;
register acc(1:16); -- accumulator;
register c;          -- carry indicator;
c:acc = add(opr,acc);
```

2) sub: this is the same as the previous element with the exception that the second operand is subtracted from the first

one. The first element of the output vector holds the borrow in, e.g.

```
c:acc = sub(opr,acc);
```

Since increasing or decreasing by one occurs quite frequently, the next two elements are incorporated for this particular purpose.

3) add1: this element takes a vector as its only input, and after increasing it by one, returns a vector of the same size, e.g.

```
register ic(1:16); -- instruction counter;
ic = add1 ic;
```

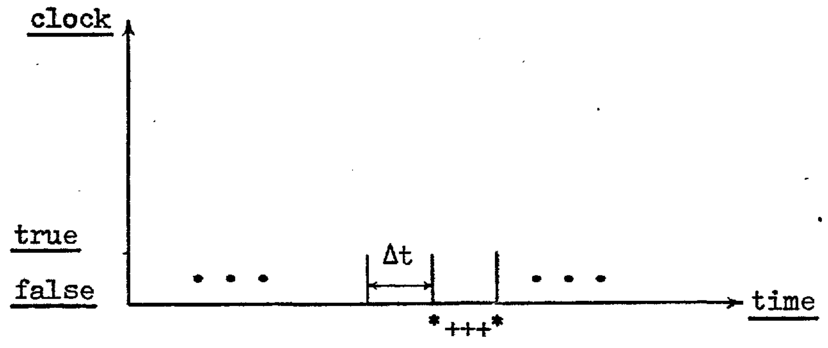
4) sub1: this is the same as the previous element except that the input vector is decreasing by one, e.g.

```
register src(1:16); -- shift count register;
src = sub1 src;
```

3.2 Bistables

3.2.1 The Requirements of Bistables in Relation to the clock Waveform

It was stated in Section IV.7.3 that in order to implement the Third Assumption a clock generating the following waveform is needed:



* assignments take place

+ expressions are evaluated

Figure 1. Ideal waveform of the clock

Two problems concerned with the clock waveform are considered here: firstly, if an expression takes longer than Δt to be evaluated and has to be evaluated at every point in time, the Second Assumption is violated. To overcome this, the user has to define his own clock to run at a slower rate; alternatively logic elements with faster switching time could be used at a higher price.

Secondly, because of certain physical constraints the clock waveform in practice would look like:

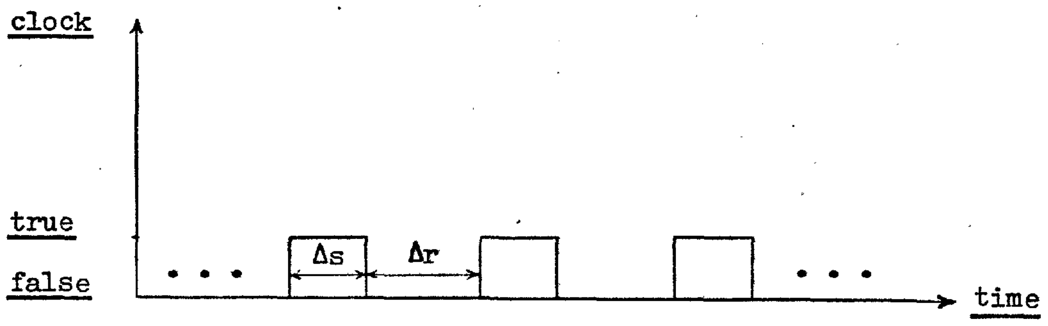


Figure 2. Actual waveform of the clock

Under this condition, there is a possibility that the contents of a register, say x, are refreshed more than once during a clock cycle - this violates the First Assumption. Such a possibility arises because the expression resulting in the value of x is fast, i.e. its evaluation is fast in comparison with Δs.

Note that in the first of the two cases considered above, the problem was due to the slow expressions, as compared with the second case in which the problem was due to the combined effect of certain physical constraints and fast expressions. As far as the clock waveform is concerned, there is no solution to the second problem. Attention is therefore directed toward the other factor involved, that is, bistables employed as registers.

Since two of the three System bistables (to be described below) take as one of their inputs an actuating signal which is synchronous with the System clock, we specify the form of this clock before going into further details:

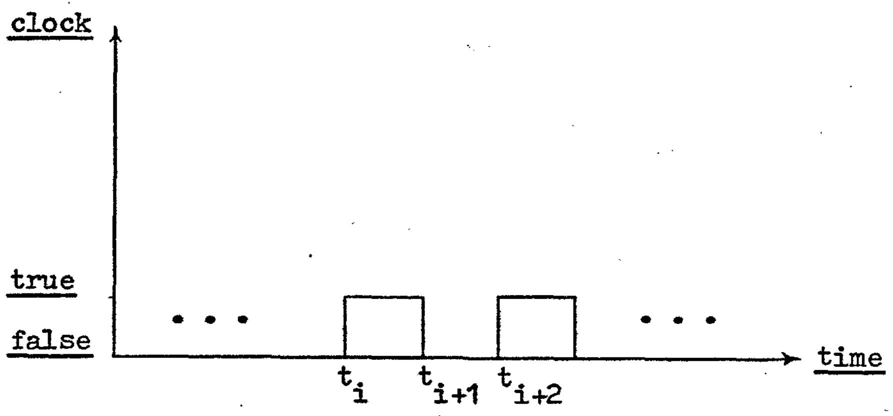


Figure 3. The System clock

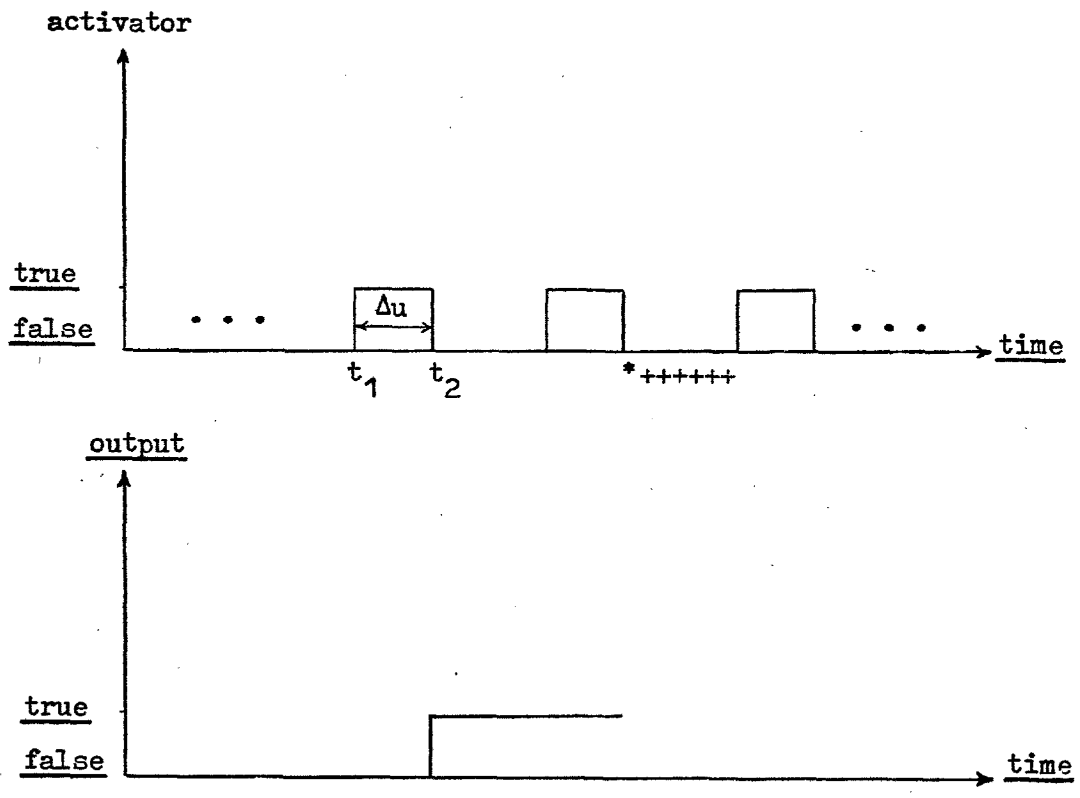
The two variables, time and clock, are defined to the System with

$$t_{i+1} = t_i + 1$$

The System initializes time to zero, and clock to false; they may be re-initialized by the user.

3.2.2 JK-bistables

One solution to the problem posed in the previous section is as follows: The storage element accepts its inputs at t_1 (Figure 4), the leading edge of the actuating signal, but the result does not appear at the output until t_2, the trailing edge of the actuating signal.



* assignments take place

+ expressions are evaluated

Figure 4. The actuating and output signals for a JK-bistable

The input signals should settle before the leading edge is reached, and they should not change while the activating signal is at the <u>true</u> level. If they do change, the output signal is indeterminate.

If q and q' represent the present and future states, the following relation should hold between them; j and k are inputs:

$$q = v(\&(\sim q, j), \&(\sim k, q))$$

During the interval Δu the bistable holds both the present and future states, although only the present state is available at the output. The JK-bistable is triggered by the trailing edge of the actuating signal.

Figure 4 shows the bistable response to j = true; k = false; it is assumed that the previous state of the bistable was false.

A JK-bistable returns a vector of two elements: the output signal and its negation, e.g.

```
Boolean q,nq,j,k;
q:nq = jkbs(j,k,clock);
```

A fourth element, which should be a binary constant, could be added to the input list so as to initialize the bistable statically, e.g.

```
q:nq = jkbs(j,k, clock, true);
```

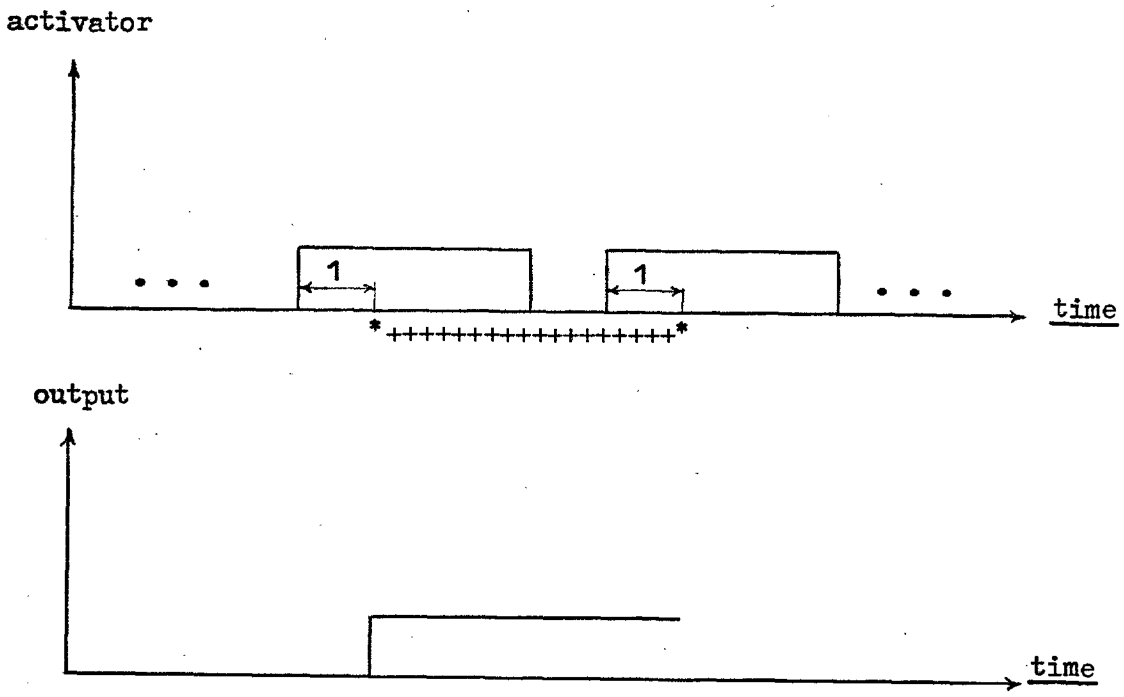
otherwise, the initial state is set randomly by the System.

3.2.3 D-bistables

Another solution to the problem raised in Section 3.1.1 is as follows. The storage element accepts its input at the leading edge of the actuating signal. The input signal is then locked out until the trailing edge of the actuating signal has passed. The following relation holds between the bistable input (d), and its future state:

$$q' = d$$

A D-bistable is triggered by the leading edge of the actuating signal. Figure 5 shows the bistable response to d = true; it is assumed that the previous state of the bistable was false.



* assignments take place

+ expressions are evaluated

Figure 5. The actuating and output signals for a D-bistable

An example of the use of a D-bistable is given below:

```
Boolean d,q;
q = dbs(d, clock);
```

A D-bistable may be initialized in the same manner as a JK-bistable, e.g.

```
q = dbs(d, clock, true);
```

3.2.4 RS-bistables

In contrast to JK- and D-, RS-bistables do not have an activator as one of their inputs, and the bistable response to a change in one of the input signals, has a delay of one unit. The following relation, presented in the form of a truth table, holds between the present and future states; r and s are inputs:

r	s	q'
false	false	q
false	true	true
true	false	false
true	true	?

The output signal is indeterminate when both inputs are at the true level; eventually the input which stays longer at this level, wins the race. Figure 6 shows the bistable response to r = false; s = true; it is assumed that the previous states of both inputs were false:

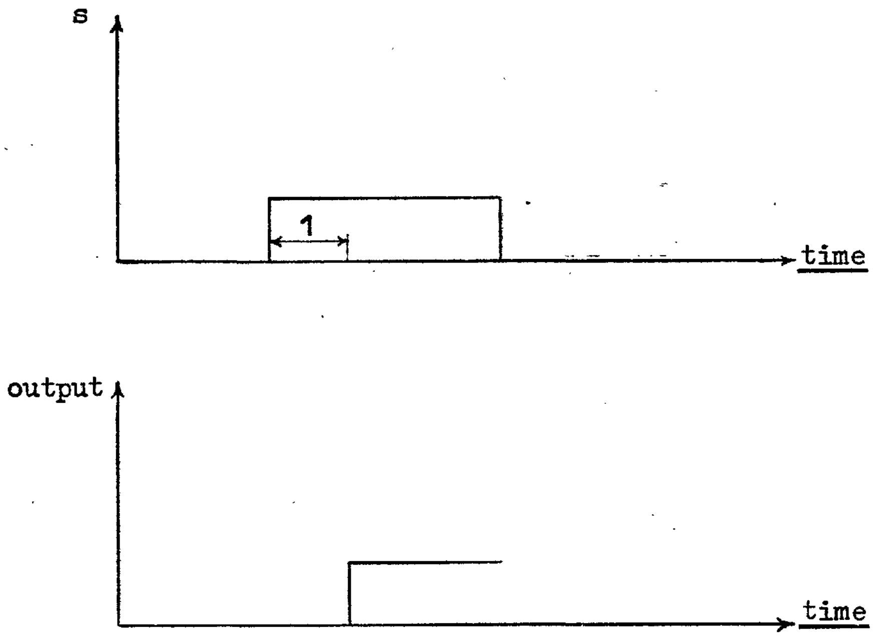


Figure 6. RS-bistable response

An RS-bistable returns a vector of two elements; the output signal and its negation, e.g.

```
        Boolean q,nq,r,s;
        q:nq = rsbs(r,s);
```

RS-bistables may be initialized in the same way as JK- and D-bistables, e.g.

```
        q:nq = rsbs(r,s, false);
```

4. Element Definition

The element definition facility helps to increase the existing capabilities of the System through defining new elements in terms of the present ones. Once an element has been defined, it is used in the same way as the standard elements of the System. In this manner, the user can extend the Language in a uniform and simple way to meet his own requirements. As an example, the majority function is defined here:

```
        element Boolean majority = (Boolean a,b,c;)
                majority = or(and(a,b), and(b,c), and(c,a));
```

In the above example, a, b, and c are formal inputs, and the output identifier, which is also the name of the defined element, is majority. An example of the use of this element in a network is given below:

```
        Boolean t,u,v,x;
        x = majority(t,u,v);
```

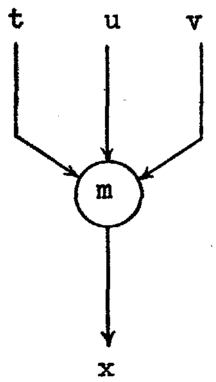


Figure 7.

As this example suggests, the syntactic form of a reference to a user-defined element is the same as that of a standard element.

A user-defined element, like a standard one, can take arrays as its inputs and return an array as its result. For example, a majority function which operates on three vectors of length 16, and returns a vector of the same size, is defined below:

```
element Boolean majority (1:16) = (Boolean a(1:16),b(1:16),c(1:16);)
           majority = or(and(a,b), and(b,c), and(c,a));
```

Elements defined by the user may impose a time delay in the course of propagating their input signals; a majority element with one unit delay could be defined as:

```
element Boolean majority = (Boolean a,b,c;)
           (delay d;
            d = or(and(a,b), and(b,c), and(c,a));
            majority = d;)
```

or alternatively,

```
element delay majority = (Boolean a,b,c;)
           majority = or(and(a,b),  and (b,c), and(c,a));
```

Similarly an element returning an array as its output, may impose a time delay on all members of the output array:

```
element Boolean majority (1:16) =(Boolean a(1:16),b(1:16),c(1:16);)
    (integer i;
     for i = 1 to 16 do
         majority(i) = dor(and(a(i),b(i)),
                           and (b(i),c(i)),
                           and (c(i),a(i)));)
```

4.1 Similarities to Sub-programs

Conceptually, certain aspects of the idea of element definition in a description language are similar to that of sub-programs in a programming language. Both cater for the construction of complex primitives from the existing ones; they thus provide the user with the facility to form a new primitive once and for all, and subsequently to be relieved of thinking about its details each time it is needed. In a programming language, this process may be repeated to adapt the language to a certain application. In a description language, at the beginning the idea serves to describe a certain technology in terms of the existing tools which, for this purpose, are regarded as abstract operators; later, in a broader context one can exploit the same idea to compose networks.

Looking from the reverse direction, that is top-down, one can decompose a problem into sub-problems of a lower degree of complexity, and thus nearer to the ones which can be handled with the aid of available primitives. This cycle may be repeated until the gap between the required function and the existing resources is filled.

In the following sections, we present two examples of function construction related to the kinds of uses mentioned above.

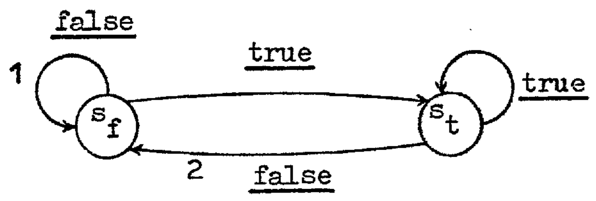
4.2 A Bistable

In this section, as an example of constructing a new primitive, we describe a bistable with the property that it changes state if its input is true, and remains unchanged otherwise. Furthermore it is required that the input signal be accepted while an actuating signal is at the false level; the input is locked out when the leading edge of the actuating signal is reached, and is not unlocked until the trailing edge is passed, at which time the bistable response to the input signal appears at its output.

If t denotes the input, and the present and future states are represented by qp and qf, the following relation should therefore hold:

$$qf = \underline{or}(\underline{and}(\underline{not}\ t,\ qp),\ \underline{and}(t,\ \underline{not}\ qp))$$

Figure 8 shows the state diagram of the bistable. There are two states, s_f and s_t, and the transition between them occurs as a result of a change in the actuating signal. The numbers written on the arcs correspond to the actions to be taken when the transitions occur.



1- accept inputs

2- transfer qf to qp

Figure 8. The state diagram of the bistable

In the following description of this element, the two possible values of variable, s, correspond to the states s_f and s_t.

```
element Boolean qp = (Boolean t, activator;)
        (Boolean s, qf;
         if nor (s, activator) then qf = or(and(not t, qp),
                                           and(t, not qp));
         if and (not s, activator) then s = true;
         if and (s, not activator) then (qp = qf; s = false;))
```

4.3 A Parallel Adder

The example presented in this section concerns network composition - a 16-bit parallel adder is constructed from half-adders. A half-adder can be described as:

```
element Boolean h-adder(c,s) = (Boolean x,y;)
      (h-adder(c) = and(x,y);
       h-adder(s) = eor(x,y);)
```

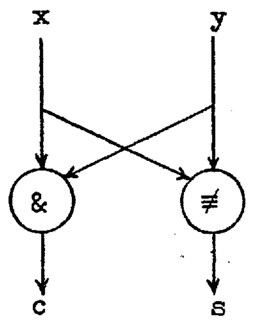


Figure 9. A half-adder

The first element of the output vector holds the carry out, and the second element holds the sum. A full-adder can now be constructed:

```
element Boolean f-adder(c,s) = (Boolean x,y,z;)
      (Boolean c1,c2,s1,s2;
       c1:s1 = h-adder(x,y);
       c2:s = h-adder(s1,z);
       c = or(c1,c2);)
```

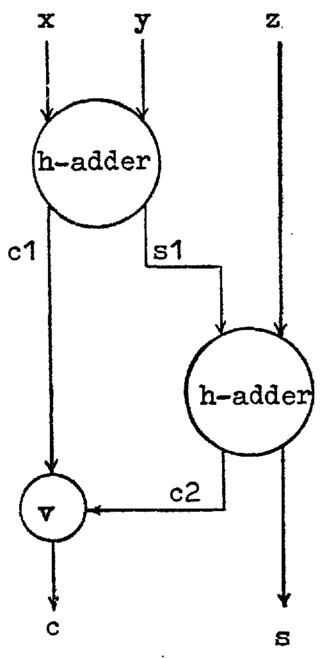


Figure 10. A full-adder

The following is a description of a 16-bit Parallel adder in terms of full-adders:

```
element Boolean s(1:16) = (Boolean x(1:16), y(1:16);)
        (Boolean c(1:17);
        c(1) = false;
        for i = 1 to 16 do
            s(i):c(+(i,1)) = f-adder(x(i), y(i), c(i));)
```

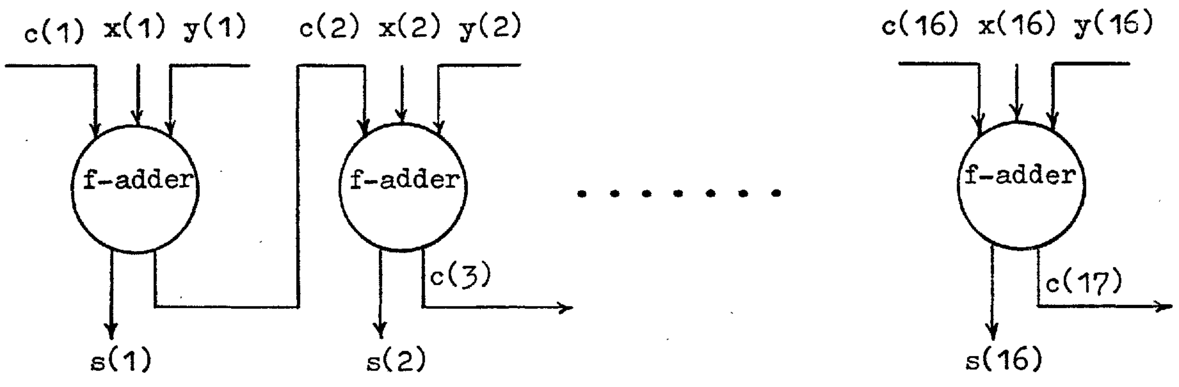


Figure 11. A 16-bit parallel-adder

Propagation delays and techniques for speeding up the carry propagation can easily be introduced and investigated.

4.4 Contrasts to Sub-programs

4.4.1 Local Variable

In a conventional program, several calls to a sub-program indicate that the same segment of code should be executed several times, possibly with different arguments. In a description, however, several references to an element mean that several instances of that element exist in the network; these instances may well be in different internal states.

Local variables of an element are the ones which do not appear in the input list, and therefore they are declared in the body of the element description. These variables may be employed to represent internal states of elements; in this respect, a helpful feature in a description language is that the values of these variables remain intact from the end of one activation to the beginning of the next one. Otherwise, in order to achieve the same effect, the user has to represent internal states by variables declared in the input list. This decreases both the clarity of descriptions, and the correspondence between symbolic and graphic descriptions.

The same argument, about the values of local variables, applies to the output variable. In the example of Section 4.2 the output variable corresponded to the present state of the bistable.

4.4.2 Linking the Main-description and Elements

The actual inputs to an instance of an element are determined by the main-description; when that instance is activated, the signal corresponding to the actual inputs should be made available to the element.

In programming languages, a call by value ensures that the value of the corresponding actual parameter is not changed by the sub-program, as contrasted to call by reference or call by name. This feature is a particularly useful one to incorporate in a description language, for if both the main-description and the user-defined element assign values to the same variable, there is the possibility of an ambiguous connectivity (Section II.7.1) being created by the user. For a similar reason it is useful to have a user-defined element as the only object which can assign a value to its output variable.

5. Simulation Facilities

Three structures, namely initialize, restart, and stop are incorporated in the System merely for the purpose of simulation. Their behaviour is similar to RT-operations, in that they need to be activated; however none of these activates its successor.

1) initialize when invoked terminates the present simulation run and starts a new one. The initializations are performed and the model is then entered. This serves to run the model for several sets of input data.

2) restart when invoked terminates the present simulation cycle, but retains the state of the model. The initializations are ignored and a new cycle is then begun. This helps when one wishes the initial state of the model to be the result of the previous run.

3) stop when invoked terminates the simulation. No new run can then be started.

6. Input/Output Facilities

The facilities described in this section are elementary, and only involve those features which have been implemented. An I/O structure behaves in the same way as an RT-operation, in that it needs to be activated so as to operate, and upon completion it activates its successor.

For input, the variables named in the input list take on values from the 'input stream'. Each data item in this stream should be separated by a semi-colon from the next one. Array names refer to the whole of the array; the same is true of sub-arrays and array segments identified by subscripted variables. As far as <u>delay</u> elements, joint and disjoint operations are concerned, the input structure acts like an RT-operation, with the variable identifier on the left hand side, and the data item to be input on the right hand side.

For output, the values of variables in the output list are transferred to the 'output stream'. A certain number of <u>spaces</u>, held in the variable <u>space</u>, are inserted after all the values corresponding to each variable are transferred. The default value of <u>space</u> is 5; it can be initialized to any decimal value by the user. As far as <u>delay</u> elements, joint and disjoint operations are concerned, the output structure acts like an RT-operation, with the variable name on the right hand side.

Among other output facilities, there are structures for inserting new-lines and spaces in the output stream.

7. <u>An Overall View of a Description</u>

In this section we shall consider the most general form of all the levels so far considered. This will give us the ability to integrate several description segments at different levels into one unit whose working is understood in terms of the general level.

One form of integration which was seen at the end of Chapter II, came as a result of interpreting the behaviour of a network described at the Level of Logic Elements in terms of a sequence of RT-operations; it resulted in the ability to combine description segments belonging to these two levels.

Another form of integration was seen in Chapter IV: since sub-levels of the Control Level were all presented in the form of controlled networks, we managed comfortably to combine them, and as a result of this aggregation the Third Assumption was introduced.

Thus in order to achieve total integration on all levels, we can now focus our attention only on the Control and Register Transfer Levels. The state of a network at the latter level is re-evaluated at every time point; in this respect, such a description is similar to that of a controlled network with a true control condition. Therefore in order to interpret, at the Control Level, the working of a network whose description is given at the Level of Logic Elements, it is sufficient to prefix the description with a true control condition; the sequence of RT-operations now takes the form of a sequential network. As was stated in Section IV.4, in this special case both the control condition and sign may be omitted. Thus if such a description is to be combined with other segments, described at the Control Level, it is sufficient to place them textually next to each other. The particular textual position is important when joint operations are involved.

It can now be said that the Control Level is the most general form of all the levels. The understanding of this general level is important from two viewpoints: firstly, it shows how several segments described at different levels interact with each other, and subsequently helps the user to understand the semantics of the notation; secondly, it forms a basis for the implementation of the Language by determining the design criterion to be the requirements of the Control Level. Having formed such a basis we are now in a position to discuss the problems involved in the implementation of the Language which takes us to Chapter VI, but first an example of the combination of description segments is presented in the next section.

7.1 An Example

The description of the serial binary counter given in Section II.9 is expanded here to include structures for initialization, simulation, and output.

```
1-    delay  cnt(1:16),b;
2-    Boolean x, c, d;
3-         cnt = false;
4-           x = true;
5-           time = 0;
6-   start;
7-          eq(time,1) → x = false;
8-          eq(time,16) → (time = 0; x = true;
                           newline; write cnt;
                           restart;)
9-          c = or(x,b);
10-         d = and(c,cnt);
11-         b = d;
12-         cnt = and(or(c,cnt), not d);
13-   end;
```

Lines 9 to 12 are the same as lines 1 to 5 in Section II.9. Lines 7 and 8 serve to produce the following waveform.

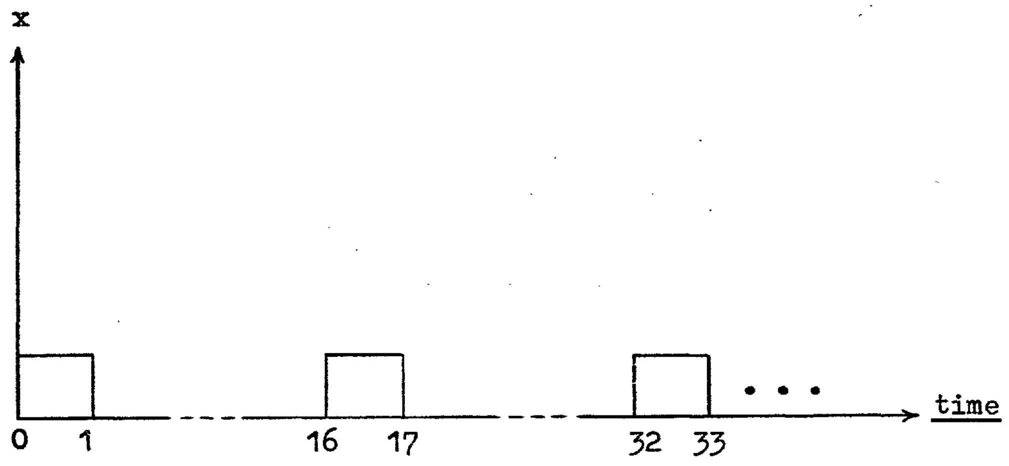


Figure 12. The signal x

The main body of the description includes two controlled networks (lines 7 and 8), with the rest represented at the Level of Logic Elements. The output obtained from the simulation run is as follows:

$$\overbrace{1000000000000000}^{16}$$

```
0100000000000000
1100000000000000
0010000000000000
       •
       •
       •
```

CHAPTER VI

The Implementation

1. Introduction

In this chapter we consider the problems involved in implementing the Language introduced so far. A detailed discussion of the implementation would be too long for inclusion here, and furthermore would cover topics - such as details of syntax and semantic analysis, administration of the symbol table, compilation of control structures - which are well discussed in the literature of translator writing; for example [28,31]. Since compiler technology is so advanced that suitable solutions exist to these problems we do not refer to them again in this chapter. Instead we discuss particular problems encountered in the implemtation of our Language and the main strategies adopted. The solutions suggested to these problems are not in general the best, but rather have been selected for reasons specific to this project which are mentioned in the following sections.

2. The Choice of the Implementation Language

Among the languages available, BCPL was particularly reported to have been used for compiler writing [5]. A persistent attempt, at the beginning of this project, to use this language for writing a compiler was unsuccessful, mainly because of the difficulties in joining segments of a program, and later in linking modules of the compiler. While writing the compiler, it was noticed that a considerable amount of code was produced merely to overcome the deficiencies of the language, and subsequently the compile time and runtime errors increased unnecessarily.

Meanwhile the author became familiar with the Translator Writing System BCL [12], and some early experiments with this language returned promising results. As compared with BCPL: the lexical analysis phase was handled by the BCL System, far less code was needed to compile the same source segment, runtime and compile time errors decreased, the language lent itself to modular programming, and the runtime was acceptable. A compiler writer's view of BCL, together with certain critical remarks and suggestions for the improvement of the language, is given in Appendix B.

3. Major Strategies

The choice of BCL was an important decision, for it dictated other strategies such as the use of a top-down parser, the method of linking the modules of the compiler, and the amount of work to be done in the syntax and semantic analysis phases.

The second decision was to make the compiler manoeuvrable so that new features could be included and tested easily, and the expansion of the compiler could be achieved in a uniform way without making major changes in the existing modules. As a result of this decision the source description was translated into an intermediate one, hereafter referred to as the object description, which was then interpreted. If the intermediate description had been translated into machine code the compiler would not have been as flexibly modifiable.

Interpretation of the object description entailed, of course, a higher runtime, proportional to the length of the object description. We were therefore led to select an order code for the object machine which would decrease the length of the object description. This in turn led to compiling the code for a zero-address machine, i.e. using a runtime stack.

Topics discussed in this chapter mainly involve those features which distinguish our Language from conventional programming languages. These are:

linking the instances of the user-defined elements with the main-description, the implementation of delay elements and delayed Boolean operators, certain aspects of the declarations, the general networks, and the for structure.

4. Linking the Modules

A 'module' is a general word for a main-description or a user-defined element. In general the description of a digital system is composed of several user-defined elements and a main-description. The main description may activate an element, and this in turn may invoke other elements. For the sake of uniformity we assume that the Description System itself activates the main-description so that the activation of this description is not different from the activation of the elements.

In general, there are two 'classes of variables' associated with a module: parameter variables and local variables; the former refers both to the input variables and the output variable. The main-description does not have any parameter variables, and therefore all those declared in it are regarded as its local variables.

Since each class of variables may contain variables of the type integer, Boolean, or delay, there are six 'address spaces' associated with a module; each space is composed of a set of addresses, starting from zero, relative to one of six base registers.

At the end of compilation six 'storage spaces' are assigned by the Description System to each instance of a module; clearly each storage space corresponds to an address space.

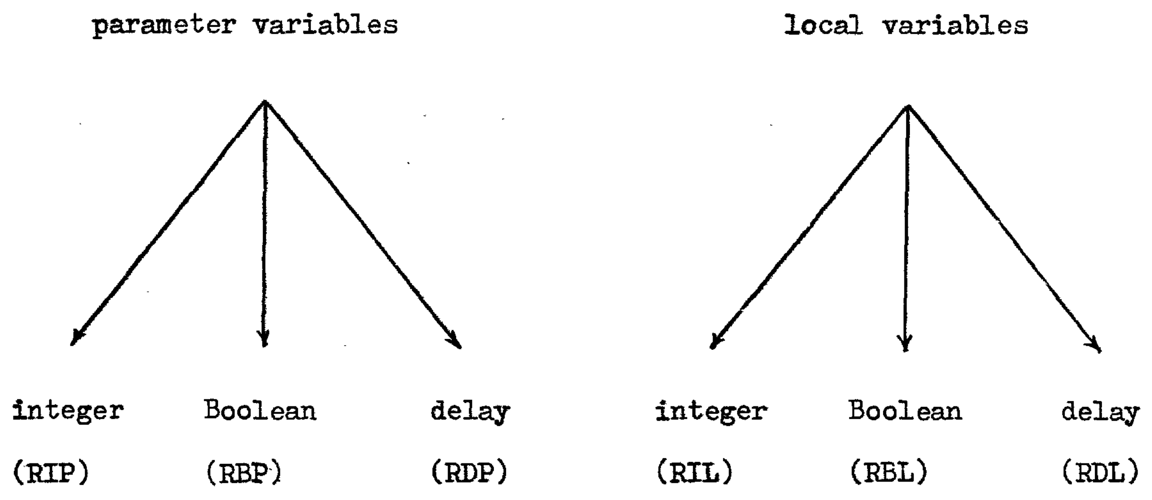


Figure 1. Address spaces and classes of variables

Once the storage requirements of each module are known, the size of the storage space for the whole description can be determined; we assume that the starting address of this storage is zero. At the start of runtime the base address of the memory data segment to be used by the description is determined by the operating system. Since the storage allocation is done on the basis that this base address is zero, all the addresses are in fact relative to this new base address which is held in a certain base register, called RDS.

When a module is activated at runtime, the activating module passes to the activated module the base addresses of the six storage spaces required. Six 'link registers' are employed to establish the link; the names of these are shown in brackets in Figure 1. As shown in this figure, a total of six categories of variables may be encountered at compile time; when a variable is then analysed, its address is considered as relative to the corresponding base register.

With the aid of this arrangement we can use the same routines for compiling the structures in a main-description or in a user-defined element.

5. Declaration of Modules

The Description System keeps a 'module declaration file'; each of its records corresponds to a module in the description and contains the following information: the name of the module, the size of each of the six address spaces associated with it, and a pointer to the beginning of the storage area where the object code for the module stands.

The name of a user-defined element is the same as its output identifier. A main-description is preceded by a directive specifying its name. The first record of the module declaration file is reserved for the main-description. At the end of compilation the file contains all the information it should have.

6. Referencing the Modules

We mentioned that when a module activates another one the base addresses of the storage spaces should be loaded into the link registers. In this section we describe a 'module reference file' which serves to determine the base addresses for the main description and instances of user-defined elements. Here we are concerned with each instance, since as mentioned in Section V.4.4.1 a digital system in general may contain several instances of a certain user-defined element, each in a different internal state.

Whenever at compile time a reference is detected to a user-defined element, a new entry is created in the module reference file by inserting the name of the element referenced and the name of the module which makes the reference. The index corresponding to this entry is then supplied to the routine analyzing the reference, and is used to load the link registers with the base addresses of the storage spaces.

As well as the name fields each record has six fields to hold the sizes of the six address spaces associated with the instance, and another field to hold the starting address of the object code corresponding to the instance. At the end of compilation the contents of these fields are set using the information already existing in the module declaration file. The routine which does this task can also ensure that all the modules referenced are declared.

The first record of the module reference file is reserved for the main-description so that the Description System can easily activate it.

7. Determination of the Base Addresses

With the aid of information so far collected in the module reference file the Description System can now determine the base addresses of the storage spaces required by each instance; these are relative to the register RDS. However, in doing so two things must be taken into consideration.

Firstly, each record of the module reference file has two fields containing the size of the storage spaces for the integer variables, and four fields containing the sizes of storage spaces for Boolean and delay variables. We assume that the sizes of the integer spaces are expressed in terms of words; each word being one unit of store. The sizes of Boolean and delay spaces, which are stated in terms of bits, must be converted into words by dividing the size by the number of bits per word. If the result is not a whole number the nearest greater number is taken as the result.

Secondly, for each module a separate storage space should be allocated for the local variables of each instance that the module refers to. Whereas for the parameter variables only one storage space is sufficient; however, the size of this space should be the maximum of the storage spaces required.

When the six base addresses for each instance are determined the Description System places them in the size fields of the record, since the sizes are no longer required.

8. The Virtual Stores

There are three types of address spaces, namely integer, Boolean, and delay spaces, which correspond to three 'virtual stores', namely integer, Boolean, and delay stores. In this section we consider how to convert a virtual address of a certain type to a real address corresponding to a 'real store'.

An integer, Boolean, or delay address is one belonging to an integer, Boolean, or delay address space. Since an integer address refers to one or more words - each word being a unit of the real store - no problem is encountered in this case; that is, the virtual store for the integers is addressed in the same way as the real store.

Each Boolean or delay address refers to one or more bits. In this respect the virtual stores for both types are addresses in the same way. There is, however, a time factor involved in the addressing of the delay store. In the following sections we investigate Boolean and delay stores.

8.1 The Boolean Store

Here the problem is how to convert a Boolean address into its corresponding real store location. The Boolean address to be converted is of course relative to one of the base registers RBP or RBL (Figure 1.), depending on whether it relates to a parameter variable or a local variable.

In order to convert the address it should be divided by the number of bits per storage word; the quotient is the displacement with respect to the base register, and the remainder indicates which bit of the word is addressed. Having determined this bit position, the Description System can retrieve or insert the information required.

8.2 The Delay Store

This store is also referred to as the 'delay line'. In this section we first describe a primitive solution to the maintenance problem of the delay store; this is then modified and improved for better efficiency.

The Description System performs a cycle for each point in time, during which the state of the digital system is re-evaluated. At the end of each cycle the contents of the delay line are shifted forward by one position, that is, for every virtual address n, the contents of the (n+1)th position after the shift is the same as that of nth position before the shift. To make this task simple the Description System uses a contiguous area of store to represent the delay line; this is in accordance with the storage allocation scheme discussed earlier.

Obviously this solution is not efficient. To improve it we assume that all the delay addresses are relative to a base address which indicates the start of the delay line. At the end of each cycle, instead of physically shifting the contents of the delay line, the Description System now simply increases the base address by one.

As the delay line base address, held in the register DBA, is increased, the locations at the beginning of the line are no longer referred to, while certain addresses refer to positions beyond the upper limit of the delay line. To overcome this difficulty, whenever a delay address is greater than the upper limit the Description System decreases it by the length of the delay line. The 'effective address' thus obtained refers to the beginning of the delay line, i.e. the positions which would otherwise not have been referred to. When the base address itself reaches the upper limit it is reset to its initial value.

This approach, although better than the previous one, has the disadvantage that every address - and at the end of each cycle the contents of DBA - have to be tested against the upper limit. The following solution is suggested to overcome this problem.

We choose the smallest number m such that 2^m is greater than or equal to the size of the delay line. The positions of the line are addressed from zero onward, and the register DBA is of m bit length. The effective address is also formed in an m-bit register. The problem of testing and decreasing the address is now solved automatically, for whenever the base address or the effective address exceeds the permissible limit an overflow occurs whose net effect in this case is that the generated address is decreased by 2^m which is what we want.

The price paid for this achievement is that the delay line is padded up so that its length is 2^m.

9. Declarations

9.1 Array Declarations

A linked linear dictionary is used in which the following information (descriptors) is kept for each variable:

1) the variable identifier
2) class: local or parameter

3) type: integer, Boolean, delay or constant
4) rank: scalar, vector, or matrix
5) the lower and upper bounds for each dimension; this applies only to vectors and matrices. If a register is described by the nested-top-down method the lower biund is assumed to be zero, and therefore the upper bound is the same as its size.
6) the base address of the virtual store space assigned to this variable.

When a declaration like:

<u>register</u> instruction (op-code(0:4), format, address(0:9));

is analyzed the above information is formed into a record for each variable; this record is then appended to the dictionary. Furthermore, the hierarchical structure presented by the declaration is set up.

For a scalar the required information is immediately available, but the size of a vector is not known until its rightmost bracket is encountered during the syntax analysis. In the above example, this implies that the declaration of the register, instruction, in the dictionary is to be postponed until the registers, op-code, format, and address, have been declared. This postponement serves to set up the hierarchical structure desired, as described below.

At compile time there are six pointers initialized to zero which correspond to the six address spaces mentioned before. After allocating a storage space to a variable the corresponding pointer is advanced. If a register is described by the nested-top-down method the value of the address pointer is saved before the sub-registers are declared in the dictionary. When the rightmost bracket of the register is encountered, i.e. when its sub-registers are declared, the register itself will be declared in the dictionary; the base address of the virtual store space for the register is what was saved at the beginning - not the current value of the address pointer.

Since the hierarchical structure of declarations is recursive, a recursive implementation language is of great help in implementing the above scheme. In the above example the order of declaring the variables is:

op-code, format, address, instruction

The fields of the dictionary records concerned with the identifier, the base address, and the link are shown below:

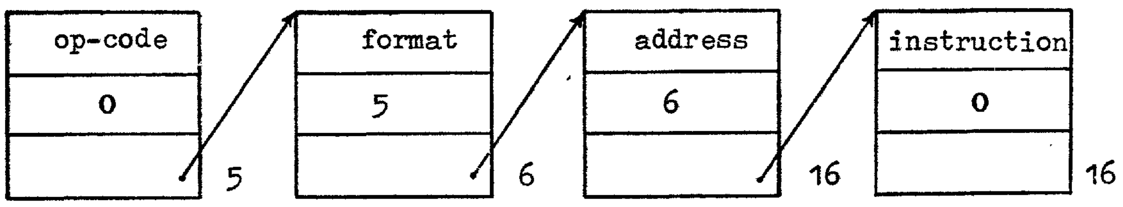


Figure 2.

ptr is the pointer corresponding to the Boolean virtual store; after declaring each variable its value is given at the right hand side. As can be seen the base addresses of the registers, instruction and op-code, are the same. The structure thus set up could be illustrated by the following figure:

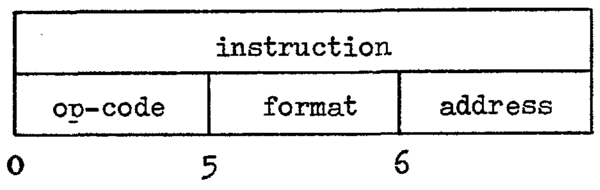


Figure 3.

9.2 Sub-array Declarations

An example of a sub-array declaration is:

```
sub-array ix(0:1) = address(1:2);
```

The analysis of the right hand side variable determines its type and base address, which are the same as those of the left hand side.

Also the rank and the lower and upper bounds for each dimension must be the same for both sides. Having obtained the required information, a record is formed for the register, ix, to be appended to the dictionary.

10. The System Body

As was seen in Chapter IV the body of a system in general begins with certain initializations which are followed by a general network. The initializations take place first; a cycle involving two steps is then repeatedly performed over the general network. In the first step all the control conditions are evaluated, and in the second the networks whose control conditions are <u>true</u> are activated. The operations corresponding to such networks are performed according to their textual ordering. The object description has to be compiled in such a way that the above effects are achieved. To illustrate how this is done let the following be the body of a system:

```
            -- initializations;
    start;
            w1 → A1
            w2 → A2
            .
            .
            .
            wn → An
    end;
```

where each w_i is a control condition, and each A_i is a sequential network. The object description is equivalent to:

```
init:
            -- initializations;
start:
            $t_1 = w_1$;
            $t_2 = w_2$;
            .
            .
            .
            $t_n = w_n$;
            if $t_1$ then $A_1$
            if $t_2$ then $A_2$
            .
            .
            .
            if $t_n$ then $A_n$
            -- miscellaneous actions;
            go to start
```

where init and start are labels, and each t_i is a temporary location in the Boolean virtual store. The cycle is repeated by a branch instruction. The miscellaneous actions refer to operations such as increasing the time and the base address of the delay line.

As mentioned in Chapter IV networks without a control condition are assumed to have a true condition. An important advantage of this approach to compiling the object description is that no special preparations need be made for compiling such networks. No temporary location is needed for them, and the position of their object code in the object description corresponds to that of their source code in the source description.

The simulation structures, <u>initialize</u>; and <u>restart</u>; are simply translated into branch instructions to the lables, init and start.

11. <u>RT-operations</u>

Two stacks are used at runtime in order to simulate RT-operations. The first one, the V-stack, serves to store values of the variables involved in an RT-operation, and the second one, the D-stack, holds the descriptors of such variables.

The set of operators of the Language is extended to include the System-defined elements, and the concatenation, assignment, and subscription operators. The latter are used in the infix form, while the other operators of the Language are in prefix form.

As mentioned earlier, the source description is translated into an object description which is then interpreted. Each operator of the Language, encountered in the source description, is translated into its corresponding operator in the object description, with its operands expected to be on the top of the V-stack. At runtime, the interpretation of each operator invokes its associated runtime routine which performs the necessary actions; control is then passed to the interpreter which repeats the same cycle. For example, the runtime routine associated with the <u>not</u> operator involves two steps. Firstly, it analyses the top element of the D-stack to find out whether its operand is a scalar or an array; secondly, it negates the operand. The routine associated with the <u>and</u> operator is slightly more complex; first it determines whether its operands are both scalars, both vectors, or one a vector and the other a scalar; then the decision is taken as to what should be done. The operators are in general concerned with both stacks; the concatenation operator is, however, one which is only concerned with the D-stack.

The scheme described above has the advantage that very complex operators may be included in the object description; this simplifies the translation task while the interpreter preserves its modularity, and hence the features of the source language can be easily tested. For example, the JK-bistable is simply dealt with by loading its three operands and their descriptors onto the stacks, and then calling the appropriate runtime routine.

Parallel RT-operations are translated by postponing the assignment operation until all the operands involved in the operations are loaded into stacks.

12. Considerations Concerning the delay Element

During each time interval, each delay element has a certain output signal; furthermore, a certain input signal, which will appear at output at a later point in time, is determined for each delay element. If the input signal were determined at a Computation Time such that thereafter no reference were made to the output signal, it could enter the queue associated with the delay element, since there would be an empty position in this queue. In general this is not the case, i.e. the output signal is referred to after the input signal is determined.

This problem is overcome by allocating n + 1 positions, from the System delay line, to an n-bit delay element.

13. Delayed Boolean Operators

An RT-operation like:

```
register a,b;
b = dnot a;
```

can be translated as:

```
register a,b;
delay t;
t = not a;
b = t;
```

where t is a one unit temporary delay element.

This combination shows a special use of the delay element, in which the output is immediately referred to after the input is determined, and furthermore no reference is made to the output in other parts of the description. This feature can be exploited to translate a delayed Boolean expression in such a way that no temporary delay element is involved; the advantage being that instead of using two locations from the System delay line, only one location from the Boolean store is used.

This is achieved by associating a temporary location with the operator to hold the value of the expression for the next time interval. When the result of the expression is referred to, first the contents of this location are loaded onto the V-stack, and then the value of the expression is stored in the location to be used in the next time interval.

The amount of space saved in this manner depends on the number of delayed operators used in the description under translation.

14. Element Designators

Whenever an element designator is encountered during compilation an entry is created in the module reference file, and the index corresponding to this entry is passed to the routine in charge which creates object code for:

1) swapping the contents of the link registers with the contents of the fields holding the base addresses of the six storage spaces corresponding to the new entry

2) storing the values of the input variables in the three storage spaces corresponding to the parameter variables

3) branching to the start of the object code for the element referred to

4) repeating the action described in (1); this resets the link registers to their initial value.

The element returns its results in the V-stack, with the D-stack containing the descriptors of the result.

15. The <u>for</u> Structure

As can be seen from the syntactic form of this structure (Section IV.8.1), the initial value, the final value, and the step of the <u>for</u> index are integer constants, and therefore known at compile time.

The structure is translated by reproducing, for each value of the index, the object code corresponding to the sequential network of the <u>for</u> structure. In this manner several instances of the network are created which differ from one another only in the value of the index.

If the sequential network includes an element designator an entry is created in the module reference file for each value of the <u>for</u> index; the effect being that separate storage spaces are created for each instance. Note that the same code segment operates on all the storage spaces thus created.

16. The Implementation Work

At the beginning of this project work began with the design and implementation of the Level of Logic Elements, and a compiler was developed for this level. The concepts related to the Control and Register Transfer Levels were introduced later, and the original compiler was extended as much as possible to include the features of these levels. As this effort continued it was noticed that certain aspects of the Control and Register Transfer Levels dictated the inclusion of certain features which were fundamental to the overall design of the compiler. These are: the basic requirements of the Control Level, features for translating expressions involving vectors, and considerations related to the user-defined elements and element designators.

In the special case that the expressions involve only scalars, one of the System stacks - the V-stack - is sufficient for the simulation of the operations; therefore the object language need include operation codes operating on this stack only; this simplifies the structure of the interpreter

CHAPTER VII

Concluding Remarks

At the beginning of this thesis we set out the requirements for the Description Language. In this chapter we examine how far these requirements have been met by the Language, and subsequently review the related work which has been carried out in this area, including a summary of other design languages developed so far, the use of programming languages in description and simulation of digital systems, and the types of simulators adopted. The effect of the underlying structure of the simulator on the language features is emphasised to point out the limitations of a particular implementation.

1. An Overall View

1.1 Procedural Features

The Language incorporates the sequencing mechanism and the control structures which are common to most programming languages. This is to be expected since there is no essential difference between a high level description of a hardware component and that of a software routine performing the same task. This is clearly demonstrated in the high level description of the example given in Section IV.9.

The ability to write procedural algorithms in the Description Language also shows the correspondence to the language of flowcharts which, as mentioned in the introduction, has been used in the design of digital systems.

1.2 The Hierarchical Structure

The examples of the shift-left-and-count component (Section IV.9), and of the parallel adder (Section V.4.3), show the adaptability of the Language to top-down and bottom-up methods of design, and its suitability in suppression of details at higher levels. The first example presents seven descriptive levels of the same component; these range from the initial problem statement to the final solution, presented at the hardware level. The second example shows the use of user-defined elements in constituting descriptive levels of the Language. The close correspondence to the graphic description demonstrates the suitability of the language of block diagrams, in this class of problems, in representing levels of digital systems.

1.3 Formal Description

Appendix A presents a formal description of the IBM 1130 in the Description Language. The manufacturer's user-manual [32] has been considerably condensed, without loosing its intelligibility. The result is a concise and precise description which can efficiently replace the manual, when the user is familiar with the System architecture, and has a basic knowledge of the instructions.

One of the results of the precision of a formal description, as contrasted with a verbal description, is that the effect of not complying with the special requirements of certain instructions - such as their mode or length - is explicitly expressed.

1.4 Implementation

It was shown in Chapter VI that the newly introduced linguistic features can be conveniently implemented: the for structure is implemented by reproducing the object code; the delay elements by introducing one more level of indexing; and instances of user-defined elements by incorporating the module reference file.

Although the object code is interpreted, and no special provision is made for optimizing the code, the runtime has been acceptably low: for the example of Section V.7.1, which was presented in a complete form for processing, the total runtime for compiling the compiler, and running the description for 5000 time units, was 2 minutes; of this runtime, some 90% was spent in the processing of the compiler.

2. Other Design Languages

The design languages which have been developed so far mainly cater for one descriptive level, and are based upon certain modeling philosophies; thus, while being suitable for a certain class of problems, they cannot be effectively applied to others. Three design languages are mentioned in this section.

ISP[3] is a notation for describing the instruction repertoir of a computer, and has been used to give a formal description of the PDP 11 instruction set; it has also been extended to represent and simulate digital systems at the Register Transfer Level. The notation lacks procedural features; when the nature of a problem demands such features the user is forced to leave the notation: the authors have used programming languages and variations of flowcharts in such cases [8].

Another language [1] conceives of digital systems as being composed of two parts: the structural part, in which the operations take place, and the control part, which organises the sequencing of the operations. The structural part consists of the description of operators and their interconnection; the control part describes activators for activating operators in the structural part, conditional and unconditional jumps, and branches into parallel operations. This approach, which suits its own class of problems, is not effective when one does not wish to separate the control function from the rest of the system, as could happen at very high or very low levels.

CDL [15] is a language which is not inclined toward a particular modeling philosophy, and describes digital systems mainly as a collection of controlled networks. It has been used together with sequence charts in the design of a number of components, and in the presentation of computer organisations. The description of a computer organisation in this language involves declaration of major system components - such as decoders, memories, switches, etc.- and specification of the micro-operations forming the algorithm implemented in hardware.

3. Use of Programming Languages

Apart from the use of APL, which is mentioned in the next section, attempts have been made to extend or modify programming languages into description languages. Since the language processor already exists, this approach saves writing a new simulator; however this advantage is offset by the need to tailor the programming language for description purposes. The resulting simulator is said to be 'object code driven'.

The programming language Algol 60 has been extended by several procedures, and modifications made to its compiler, so as to simulate digital systems at the Level of Logic Elements [45]. The primitives for describing networks are logical operators, bistables in the form of system defined elements, and a one-unit delay element. These are all referred to in the form of Algol procedures, where the procedure name indicates the type of the element, and the parameters establish connectivity.

Three procedures are the constituent elements of each description: the first one determines the state of inputs at successive points in time, with the time being a global variable; the second describes the network to be simulated by calling procedures corresponding to the primitive elements; the third is concerned with the terminal whose condition is to be monitored during the simulation run.

The textual position of statements forming the description is immaterial, at the cost that at each point in time the processor scans the whole structurc of the network to form a linear description.

Another attempt to extend a programming language uses a dialect of Algol 60 for the behavioural description and simulation of digital systems [42]; the language itself, like many other programming languages, lacks hardly anything in this respect.

4. Formal Description

A description language has been proposed to describe digital systems, mainly at the Level of Logic Elements [53]. The language is suitable only for description purposes, and does not lend itself to implementation. Since the language is unaware of the concept of time, the author limits himself to open trees so that ambiguous situations do not arise.

Programming languages could be used to describe existing systems in a formal fashion: an APL description of the IBM 360 has been presented [22]; however it has been pointed out that the description, although an impressive work, is not easy to follow [52].

5. Simulation

So far we have mentioned the use of programming languages for simulation at the behavioural level, and at the Level of Logic Elements. In this section the basic structure of simulators, particularly developed for the latter level, is studied in relation to the overlying language.

One approach toward system description at the Level of Logic Elements is, for each gate in the system, to write one line of description composed of a number denoting the gate type, a second indicating the gate output, followed by several numbers indicating the gate inputs [36,40]. The resulting medium of representation, which cannot be called a language, is improved by replacing numbers by mnemonics [29]. It is usually assumed that each gate has a one-unit time delay.

The core of the simulator is a table which has one entry for each gate: its type, inputs, and output. The resulting simulator is therefore said to be 'table driven'. At the start of the simulation run, the gates connected to the inputs of the system propagate. Those gates whose input signals are changed as a result of this propogation, are scheduled to propagate at the next point in time. This process continues until a time point is reached where the user wishes to stop the system.

Apart from the obvious simplicity of implementing the table driven scheme, its main advantage is that the textual ordering of description lines is immaterial; its disadvantage is that the basic structure of the simulator is not sufficiently powerful and flexible to be effective for higher level descriptions.

The representation form, described earlier, is improved by representing each gate in the system by an assignment statement; the output identifier appears on the left hand side and the gate name, followed by input identifiers enclosed in brackets, is written on the right hand side [23]. Further improvements have been achieved by allowing prefix expressions on the right hand side, and by using the <u>for</u> statement for representing repetitive constructions [54]. User-defined elements and Boolean vectors have also been incorporated [23].

This is the limit which the language can reach on the basis of a table driven simulator. Therefore the scheme is suitable mainly for the Level of Logic Elements, and partly for the Register Transfer Level; it cannot deal with control structures, which are essential to the functional description of digital systems.

Chapter II of this thesis provides a theoretical basis and a suitable notation for presenting digital systems at the Level of Logic Elements; the notation lends itself to the table driven implementation scheme.

6. Scheduled Event Simulators

In scheduled event simulators, an event is identified by an instruction pointer referring to the start of the event, and by a time point at which the event is to be carried out.

A table of events is maintained which contains an ascending list of time points, each of which refers to a sequence of events to be executed at that particular point. Execution of an event may cause other events to be scheduled at later points in time. The simulator begins by executing the events of the first time point, and at the completion transfers the control to the next point. This process is repeated until either there is no further event to be executed, or a user command halts the simulation run.

This kind of underlying structure requires the system description to be dynamic, that is, instead of a system being described as a set of components and their interconnection, the propagation of change of signals at input terminals must be described, thus weakening the correspondence between the system and its model. This disadvantage, and the poor efficiency caused by the administrative effort required to maintain the table of events, were the main reasons for abandoning this approach when a description language and its processor were developed at the beginning of this project [50].

7. Design Automation

Efforts have been made to automate the design, e.g. [25,48], that is, given a high level description of a digital system - which is mainly a functional description - to produce the system description for implementation. Among these the most serious attempt is [24] in which the high level description is presented in a notation based on APL [34]. The implementation description is presented by block diagrams. Designing the major parts of the IBM 1800 using this approach resulted in 160% more gates than the actual design. Modifications were suggested which would decrease the number of gates to only 33% more than the actual design.

APPENDIX A

Description of a Computer

1. Introduction

In this appendix we give a user-oriented description of an actual computer at the instruction level. An already existing computer has been considered, instead of devising an example, so as better to put the Language to test.

The machine described here is a subset of the IBM 1130, i.e. the special instructions for handling interrupts and I/O operations are excluded. It has been chosen because of its limited number of operation codes (22 in the subset), which however include a variety of functions, some of them very integrated. This unavoidably results in an operation code having several meanings, depending on the modifier bits used. Since these interpretations do not usually have a common basis, it is difficult for the programmer to get used to the instructions, and therefore he often needs to refer to the reference manual. Because of this (unnecessary) complexity, the IBM 1130 seems to be a good case against which to test the Language.

In the following descriptions, verbal explanations should be treated as comments.

2. General Description

The IBM 1130 is a 16-bit word machine. There are three index registers which occupy words 1 to 3 of the main memory, and an instruction address register which in general contains the address of the next instruction to be executed.

```
memory matrix m(0:32767,0:15);
sub-array xr(1:3,0:15) = memory(1:3,);
register iar(0:15);
```

At most two consecutive words can be joined together to form a unit to hold instructions or data items. A long instruction is composed of two words, a short instruction of one.

```
Boolean double-word(0:31);
sub-array lins(0:31) = double-word;
sub-array sins(0:15) = double-word(0:15);
```

The first 8 bits are common between short or long instructions. The first 5 bits of the common part indicate the operation code, the next bit signifies whether the instruction is long or short, and the last two (tag bits) point to the register whose contents are to be used during effective address generation.

```
sub-array cpart(0:7) = double-word(0:7);
sub-array opcode(0:4) = cpart(0:4),
          f = cpart(5),tag(1:2) = cpart(6:7);
```

In shift instructions, which are always short, bits 8 and 9 indicate the type of shift operation, e.g. shift to the right, to the left, etc.

```
sub-array sind(1:2) = sins(8:9);
```

In general the last 8 bits of a short instruction specify the displacement with respect to a base address during effective address generation. The base address is contained in the register indicated by the tag bits.

```
sub-array disp(1:8) = sins(8:15);
```

In general, bit 8 of a long instruction indicates the mode of addressing: if it is true addressing is indirect; otherwise it is direct. The modifier bits (9 to 15) are employed in some I/O instructions. The address part of a long instruction occupies the whole of its second word.

```
sub-array ia = lins(8),modifier(1:7) = lins(9:15);
sub-array address(0:15) = lins(16:31);
```

The register, ov, is used in some arithmetic operations to indicate an overflow. Arithmetic operations take place in the whole or part of a register, called cacxt, which has as its sub-registers, cy for carry indication, and acxt. The latter is composed of an accumulator, acc, and an extension, ext, which is the low-order extension of acc in double word arithmetic operations.

```
register ov;
register cacxt(cy,acxt(acc(0:15),ext(0:15)));
```

3. Effective Address Generation

The effective address is formed in register ea(0:15); and is interpreted as an unsigned binary number. If the instruction is short, the displacement is expanded to 16 bits by duplicating its leftmost bit, thereby preserving its sign; the expanded displacement is then added to the contents of iar or the appropriate index register according to the tag bits.

In long instructions, after indexing, the ia bit is checked to determine the mode of addressing; if it is true, ea is regarded as a pointer to the effective address. Note that in a long instruction tag bits of zero specify no indexing, i.e. the addressing is absolute.

```
register edisp(0:15); -- expanded displacement;
if f then (if nor tag then ea = address;
                       else ea = add(address,xr(tag,));
           if ia then ea = m(ea,);)
     else (edisp(0:7) = disp(1);edisp(8:15) = disp(1:8);
           if nor tag then ea = add (edisp,iar);
                      else ea = add (edisp,xr(tag,));)
```

4. Instructions

The action to be performed by an instruction is mainly determined by the value contained in the sub-register opcode. There are 22 operation codes as defined below:

```
constant  ld = 1, ldd = 2, sto = 3, std = 4,    -- load and store;
          ldx = 5, stx = 6, lds = 7, sts = 8;
constant  a = 9, ad = 10, s = 11, sd = 12, m = 13,  -- arithmetic;
          d = 14;
constant  land = 15, lor = 16, leor = 17;           -- logical;
constant  shl = 18, shr = 19;                       -- shift;
constant  bsi = 20, bsc = 21, mdx = 22;             -- branch;
```

A case structure embraces the description of instructions:

```
case  opcode of [
      ld  → . . .
      ldd → . . .
      .
      .
      mdx → . . .]
```

In the following sections we replace the dotted lines in the <u>case</u> structure as the action performed by each instructions is described.

4.1 <u>Load and Store Instructions</u>

Load: the contents of the memory location pointed to by ea replace the contents of acc.

```
ld → acc = m(ea,);
```

Load double: the contents of the memory location pointed to by ea and of its successor replace the contents of acc and ext respectively.

```
ldd → acxt = m(ea,): m(add1 ea,);
```

Store: the contents of the memory location pointed to by ea are replaced by the contents of acc.

```
sto → m(ea,) = acc;
```

Store double: the contents of the memory location pointed by ea and of its successor are replaced by the contents of acc and ext respectively.

```
std → m(ea,): m(add1 ea,) = acc:ext;
```

Load index: a value is stored in the index register indicated by the tag bits. In a long instruction the value is contained in the address field or the memory location pointed to by this field depending on the mode of addressing. In a short instruction the expected value is the displacement.

```
register val(0:15);
```

```
ldx → (if f then (if ia then val = m(address,);
                         else val = address;)
           else (val(0:7) = disp(1); val(8:15) = disp(1:8);)
       if nor tag then iar = val;
                  else ix(tag,) = val;)
```

Store index: the contents of the index register specified by the tag bits are stored in a memory location. In a long instruction the location is pointed to by the address field, or the contents of the address field, depending on the mode of addressing. In a short instruction the location is determined by adding the contents of iar and the expanded displacement.

```
register  loc(0:15);
stx → (if f then (if ia then loc = m(address,);
                         else loc = address;)
           else (edisp(0:7) = disp(1); edisp(8:15) = disp(1:8);
                 loc = add(edisp, iar);)
       if nor tag then m(loc,) = iar;
                  else m(loc,) = ix(tag,);)
```

Store status: the status of carry and overflow indicators are stored in bits 14 and 15, respectively, of the addressed memory location. Bits 0 to 7 of this location remain unchanged, and bits 8 to 13 are reset to false.

```
sts → [m(ea,14) = cy; m(ea,15) = ov; m(ea,8:13) = false;]
```

Load status: the status of carry and overflow indicators are set according to the status of bits 14 and 15 of this instruction.

```
lds → [cy = sins(14); ov = sins(15);]
```

4.2 Arithmetic Instructions

Negative numbers are represented in two's complement.

In arithmetic instructions two registers t_1 and t_2 are used to hold the sign bits of operands for the overflow test.

Add: the contents of acc and the memory location pointed to by ea are added together and the result is placed in acc. The carry indicator is set if there is a carry out of the leftmost position, and the overflow indicator is set if the result exceeds the capacity of the accumulator.

```
a → ($t_1$ = acc(0); $t_2$ = m(ea,1); cy:acc = add(acc,m(ea,));
      if and (eqv($t_1$,$t_2$), neq($t_1$,acc(0))) then ov = on;)
```

Add double: the contents of the memory location addressed by ea and of its successor are added to the contents of acc and ext. The carry and overflow indicators are affected as in the previous instruction.

```
ad → ($t_1$ = acc(0); $t_2$ = m(ea,1);
        cacxt = add(acxt,m(ea,): m(add1 ea));
        if and (eqv($t_1$,$t_2$), neq($t_1$,acc(0))) then ov = on;)
```

Subtract: the contents of the memory location pointed to by ea are subtracted from the contents of acc. The carry indicator is set if there is a borrow at the leftmost position, and the overflow indicator is set if the result exceeds the capacity of accumulator.

```
s  → ($t_1$ = acc(0); $t_2$ = m(ea,1); cy:acc = sub(acc,m(ea,));
        if and (neq($t_1$,$t_2$), neq($t_1$,acc(0))) then ov = on;)
```

Subtract double: the contents of the memory location pointed to by ea and of its successor are subtracted from the contents of acc and ext. The carry and overflow indicators are affected as in the case of the previous instruction.

```
sd → (t1 = acc(0); t2 = m(ea,1);
     cacxt = sub(acxt, m(ea,): m(add1 ea,));
     if and (neq(t1, t2), neq(t1, acc(0))) then ov = on;)
```

Multiply: the contents of acc and the word at ea are multiplied together and placed in acxt.

```
m →  acxt = bln(mult(int acc, int m(ea,)));
```

Divide: the contents of acxt are divided by the contents of the word at ea; the quotient is placed in acc and the remainder in ext. The overflow indicator is set if the quotient exceeds the capacity of the accumulator; otherwise it will remain unchanged

```
Integer quotient;
d →  (quotient  = div(int acxt, int m(ea,));
      acc = bln quotient;
      ext = bln (rem(int acxt, int m(ea,)));
      if or (gt(quotient, 32767), lt(quotient, -32768)) then ov = on;)
```

4.3 Logical Instructions

```
land → acc = and(acc,m(ea,)); -- logical and;
lor →  acc = or(acc,m(ea,));  -- logical or;
leor → acc = eor(acc,m(ea,)); -- logical exclusive or;
```

4.4 Shift Instructions

Except for the shift left and count instruction, the number of positions shifted is controlled by the field specified by the tag bits. Each of the four shift left and three shift right instructions are defined by bits 8 and 9 (sind) of the instruction. In the following description, the first case statement is for shift left and the second one is for shift right instructions.

```
        register scount(1:6);  -- count of shifts;
        shl,shr → (if nor tag then scount = disp(3:8);
                                  -- six rightmost bits of displacement;
                                    else scount = xr(tag,10:15);
                                  -- six rightmost bits of the index register;

        case opcode of [
           shl →  case sind of [
                                .
                                .
                                .
           shr →  case sind of [
                                .
                                .
                                .
                                ])
```

4.4.1 Shift Left Instructions

```
        constant sla = 0, slt = 1, slca = 2, slc = 3;
        sla  → cy:acc = lshift(cy:acc, scount);  --  shift left accumulator;
        slt  → cacxt =   lshift(cacxt, scount);  --  shift left cacxt;
```

Shift left accumulator and count: when the tag bits are zero this instruction is executed in the same way as shift left accumulator. In any other case acc is shifted to the left until either its leftmost bit is true, or the shift count is zero. The count is decreased by one at each step, and at the end it is loaded into the six rightmost bits of the index register indicated by the tag bits.

```
        slca → if nor tag then cy:acc = lshift(cy:acc,scount);
                         else (while and(not acc(0), or scount) do [
                                      cy:acc = lshift (cy:acc);
                                      scount = sub1 scount;]
                                      xr(tag,10:15) = scount;)
```

Shift left and count acc and ext: this instruction is the same as the previous one, except that both acc and ext are shifted.

```
slc→ if nor tag then cacxt = lshift cacxt;
                   else (while and(not acc(0), or scount) do [
                              cacxt = lshift(cacxt, scount);
                              scount = subl scount;]
                              xr(tag,10:15) = scount;)
```

4.4.2 Shift Right Instructions

```
constant sra = 1, srt = 2, rte = 3;
sra→acc = rshift acc;   -- shift right accumulator;
srt→acxt = rshift acxt; -- shift right accumulator and extension;
rte→acxt = rcirc acxt;  -- circulate right acc and ext;
```

4.5 Branch Instructions

4.5.1 Branch or Skip on Condition

Six separate conditions of acc can be tested by placing a true bit in the appropriate bit position of the instruction. The bit positions, corresponding conditions, and expressions describing them are given below:

bit position	condition	expression
15	overflow indicator off	c(1) = and(not ov,sins(15));
14	carry indicator off	c(2) = and(not cy,sins(14));
13	acc contents even	c(3) = and(not acc(15),sins(13));
12	acc contents positive, not zero	c(4) = and(gt(int acc,0),sins(12));
11	acc negative	c(5) = and(acc(0), sins (11));
10	acc zero	c(6) = and(nor acc, sins(10));

Long format: When none of the conditions specified is true, the program branches to ea; otherwise the next instruction in sequence is obeyed.

Short format: If any of the conditions specified is true, the next word in memory is skipped and the second one is regarded as the instruction to be obeyed; otherwise the next instruction is executed. The overflow indicator is reset when tested by the bsc instruction.

```
bsc  → (if f then (if nor c then iar = ea;)
               else if or c then iar = addl iar;
           if sins(15) then ov = off;)
```

4.5.2 Branch and Store iar

Long format: the long bsi instruction is exactly the same as the long bsc instruction.

Short format: the contents of iar are stored in the memory location pointed to by ea; the iar is then set to addl ea.

```
bsi →  if f then (if nor c then iar = ea;
                    if sins(15) then ov = off;)
                else [m(ea,) = iar; iar = add1  ea;]
```

4.5.3 Modify Index and Skip

This instruction can be used to modify an index register, the iar, or the contents of a word in memory. Except in the case of the modification of iar in short format, a skip occurs if the index register or the memory word being modified changes sign or becomes zero. This causes the next memory word in sequence to be skipped over.

Long format: modification is accomplished according to the tag and ia fields of the instruction. If the tag bits are false the expanded displacement is added to the contents of the memory location specified by the address field of the instruction. Otherwise, the ia bit determines whether the contents of the memory location pointed to by the address field, or the contents of the address field should be added to the indicated index register.

Short format: the expanded displacement is added to the register specified by the tag bits of the instruction.

```
Boolean sb;   — sign before modification;
Boolean sa;   — sign after modification;
Boolean zero; — zero test indicator for the modified word;
edisp(0:7) = disp(1); edisp(8:15) = disp(1:8);
if f then (if nor tag then (sb = m(address,0);
                            m(address,) = add (m(address,),edisp);
                            sa = m(address,0),;
                            zero = nor m(address,);)
                      else (sb = xt(tag,0);
                            if ia then
                               xr(tag,) = add(xr(tag,),m(address,));
                                  else
                               xr(tag,) = add(xr(tag,),address);
                               sa = xr(tag,0);
                               zero = nor xr(tag,);))
       else if nor tag then (iar = add (iar, edisp);
                               — provisions for no skip;
```

```
                    sb = false;
                    sa = false;
                    zero = false;)
               else (sb = xr(tag,0);
                    xr(tag,) = add(xr(tag,), edisp);
                    sa = xr(tag,0);
                    zero = nor xr(tag,);)
```

The skip occurs as shown below:

```
if or (zero, neq(sb,sa)) then iar = add1 iar;
```

5. The System

The system itself can now be described:

```
switch start-on;
  → rest of declarations:

start;
  start-on → (-- fetch the instruction pointed to by iar;
             double-word(0:15) = m(iar,); iar = add1 iar;
             if f then (double-word (16:31) = m(iar,);
                        iar = add1 iar;)
             -- descriptions for generating the effective address;
             -- interpret the operation code;
             case op-code of [
                             .
                             .
                             .])
end;
```

6. Conclusion

Some 90 pages of the manufacturer's reference manual [32], describing the instruction set, has been condensed by a factor of six, with no loss of precision. It is worth pinpointing some of the areas where we have gained over verbal explanation. To this end we look at overflow setting in arithmetic instructions, and testing conditions in the Branch or Skip on Condition instruction.

The overflow setting in arithmetic instructions, add, subtract, and divide, is such that at the end of a sequence of such calculations, possibly related to the evaluation of an arithmetic expression, if the indicator is _on_ it means that an overflow has occurred in one of the calculations. The system achieves this effect by setting the indicator, ov, if an overflow occurs as a result of the execution of any of the three instructions mentioned above, leaving the indicator unchanged otherwise. The way this is described in our notation is:

```
        if x then ov = on;
```

where x represents the condition for overflow. Equally, one could write:

```
        ov = or(ov, x);
```

We have only expressed the condition for setting ov; the structure itself implies that the indicator is left unchanged otherwise, while in a verbal description the case in which the indicator remains unchanged has to be emphasised.

The second example is concerned with the bsc instruction. Considering the long format, a verbal explanation should describe three cases:

1) when none of the conditions is <u>true</u>

2) if any of the conditions is <u>true</u>

3) if no condition is specified

After some analysis one can satisfy oneself that the consequence of (1) and (3) should be the same, but this is not quite clear from the beginning. On the other hand, a formal description treats both cases in a unified manner namely:

```
if nor c then iar = ea;
```

It is interesting to note that in both examples a formal description is more precise and more concise than a verbal explanation, and for this reason it is easier for the programmer to retain it mentally and for the logic designer to implement it, and furthermore less examples are required to clarify it. The bulk of saving, as compared with the reference manual, is in exclusion of examples.

APPENDIX B

The Implementation Language

1. Introduction

This appendix describes the main features of BCL [12] and evaluates its effectiveness for compiler writing. Suggestions made for improvement are intended to be compatible with the present System, from both notational and implementation viewpoints.

The notation used here for presenting the language, is chosen for its suitability for reference and publication. Slight differences exist as compared with the hardware language, for example in the type identifier and in the representation of recursive groups. The language user is therefore advised to consult appropriate manuals for writing programs in BCL.

The input data to a BCL program is regarded as a string of characters. The normal mode of operation is input, in which the program occasionally tries to recognise a sub-string. The programmer can easily switch the mode to output so as to transfer information to the environment. An input pointer, called chpt, points to the beginning of the unprocessed sub-string of the input string. At the start, it is initialised to point to the first character at input. During the program execution, the pointer may move forward or backward, as will be seen later.

2. General Description

General parallelism exists between BCL and Backus-Naur Form (BNF). In BNF a meta-variable, x, is defined by a 'production':

$$x \ ::= y_{11}\ y_{12}\cdots y_{1n_1} \mid y_{21}\ y_{22}\cdots y_{2n_2} \mid\cdots\mid y_{m1}\ y_{m2}\ \cdots y_{mn_m}$$

where each y is one of the following:

1) a terminal constant, such as a keyword or delimiter
2) a terminal variable, such as an identifier or an integer
3) a non-terminal, in which case it should be defined subsequently through other productions.

A BCL description of such a production for syntax analysis purposes is given by the following 'group':

$$\begin{aligned} x\ \underline{\text{is}}\ (\underline{\text{either}}\ & y_{11}, y_{12}, \ldots, y_{1n_1} \\ \underline{\text{or}}\quad & y_{21}, y_{22}, \ldots, y_{2n_2} \\ & \vdots \\ \underline{\text{or}}\quad & y_{m1}, y_{m2}, \ldots, y_{mn_m}\ ) \end{aligned}$$

If a meta-variable, x, is defined directly or indirectly in terms of itself, the corresponding group definition should be preceeded by the word <u>recursive</u>.

As an example, suppose in a language a Boolean expression is defined to be:

```
<Boolean expression> ::= <Boolean operator>(<operand list>)|
                              <simple expression>
<Boolean operator> ::= and|or|not
<operand list> ::= <Boolean expression>{,<Boolean expression>}*
<simple expression> ::= true|false|<Boolean variable>
<Boolean variable> ::= <subscripted variable>|<simple variable>
<subscripted variable>  ::= <name>(<subscript>)
<simple variable> ::= <name>
```

This grammar generates sentences which are Boolean expressions in the language under consideration. A BCL program which recognizes such sentences is given below. Terminal constants are enclosed in quotation marks.

```
recursive Boolean expression is (either Boolean operator,
                                        '(', operand list,')'
                                 or     simple expression)
Boolean operator is (either 'and'
                     or     'or'
                     or     'not')
recursive operand list is (Boolean expression,
                           repeat(',', Boolean expression))
simple expression is (either 'true'
                      or     'false'
                      or     Boolean variable)
Boolean variable is (either subscripted variable
                     or     simple variable)
subscripted variable is (name,'(', subscript,')')
simple variable is (name)
```

The BCL notation, <u>repeat</u> (y), indicates zero or more occurrences of a meta-variable, y, as does the {y}* notation in BNF. The above description is simple in that it does not recognize syntax errors; nor has any semantic analysis been introduced; and the use of spaces and newlines in the input Boolean expressions is not allowed. We shall later demonstrate ways of improving these shortcomings.

The two meta-variables, name and subscript, are not defined any further. As mentioned earlier, when defining a language there exist two kinds of terminal variables, namely identifiers and integers. Corresponding to these, BCL introduces two types of variables - <u>identifier</u> and <u>integer</u>.

If a name is assigned either of the two types in a description, it is understood to be a terminal variable, and therefore not required to be defined any further. The maximum character length of an identifier should be given at the time of its declaration. For example, the declarations for the Boolean expression program could be:

```
     dec is (identifier name (8), :: name is an identifier of
                                  :: maximum 8 character length
               integer subscript  :: subscript is an integer variable)
```

An identifier variable matches a string of alphanumerics, starting with a letter. As a result of a successful match, the variable takes on the string as its value. The only operation allowed on identifier variables is comparison (=).

An integer variable matches a sequence of decimal digits, and a successful match results in the decimal value of the sequence being assigned to the variable. Relational and arithmetic operators can be employed to form expressions over integer variables.

3. The BCL Sequencing Mechanism

In order to obtain a better insight into the language, and to be able to process semantics, as well as syntax, we study the sequencing mechanism of BCL, i.e. the movement of the program pointer.

A BCL 'primitive unit' returns a logical value true or false each time executed. These values are not available to the programmer, and are merely used to describe the sequencing technique of the System. It is assumed here that the mode of operation is input, in which some of the primitives affect the input pointer chpt. A primitive unit is one of the following:

a) a terminal constant or variable, e.g.

'and', '(', 'true'	*(terminal constants)
subscript	+(integer terminal variable)
name	+(identifier terminal variable)

The System tries to find a match for the string constant* or the terminal variable+. A 'successful' match results in the value true being returned, and chpt being advanced to point to the first character after the matched sub-string. In the case of 'failure', the value returned is false and chpt is affected as described in Section 3.1.1.

b) An assignment statement, e.g.

```
i = j+1
a = 'begin'
```

The assignment takes place as specified; the value returned is always true, and chpt remains unchanged.

c) A conditional statement, e.g.

```
if  i < j
if  a = 'begin'
```

The value returned is that of the Boolean expression following if, and chpt is affected as described in Section 3.1.1.

In output mode, case (a) simply outputs the value of the primitive involved, instead of attempting to find a match; the logical value returned is always true, and chpt always remains unaffected. The behaviour of cases (b) and (c) is independent of the operation mode.

3.1 Units

A 'unit', which is either a compound alternative or a group, is composed of several primitive units, and the way it affects the System depends on its constituent elements. Like primitives, units return a logical value when executed. In the following sections, after describing alternatives, we shall study the effect of units on the input and program pointers.

3.1.1 Alternatives

An 'alternative' could be one of the following:

1) either 'and', '(', operand list, ')'
2) or 'false'

The program pointer moves along an alternative as long as the value returned by its primitives is true. The first false value causes chpt to be moved back to its position at the time the alternative was entered. The logical value of an alternative is true provided the value returned by each of its primitives is true, in which case the alternative is said to have been successful. Failure of any element causes the alternative to be failed.

3.1.2 Compound Alternatives

A 'compound alternative' is composed of several alternatives enclosed in brackets, e.g.

```
(either  'and'
 or      'or'
 or      'not')
```

When the compound is entered, the first alternative is scanned; in the case of a failure, the program pointer moves to the next alternative. This process continues until either a successful alternative is found, in which case the compound is said to have been successful, or, the last alternative fails; this failure causes chpt and the program pointer to be moved back to their original positions at the time the compound was entered.

3.1.3 Groups

An example of a 'group' is:

```
simple expression is (either 'true'
                      or     'false'
                      or     Boolean variable)
```

A call to a group is indicated by the occurence of the group-name, and returns a logical value corresponding to the success or failure of the group body; the input and program pointers are affected accordingly. Therefore, group calls (and similarly, compound alternatives) can be treated as primitive units, and they may be combined in a nested or linear fashion. An example of the use of groups follows.

In extended BNF, we can rewrite the productions for Boolean variable and simple variable to save some back-tracking:

```
<Boolean variable>  ::= <name>{(<subscript>)|<empty>}
```

where braces are meta-symbols of factorization, and <name> is being factorized. The corresponding BCL definition would be:

```
Boolean variable is (name,(either '(',subscript, ')'
                           or     nil.))
```

Here the primitive name is followed by a compound alternative. The primitive nil in the second alternative of the compound corresponds to the meta-variable<empty>; it is always successful and has no effect on chpt. Similarly, in the group operand list (Section 2), the primitive ',' is preceeded and followed by calls to the group, Boolean expression.

A BCL program consists of a sequence of group definitions, followed by the directive:

```
*enter (<group-name>)
```

indicating which group the program is to enter first.

4. Format Handling

Three System-defined variables are used to match spaces and end of record (EOR) markers in input data; sp. matches only one space, while osp. matches an optional number of spaces - żero or more - and nl. matches an EOR marker. For instance a group, called spc, may be defined which matches an optional number of spaces, possibly containing a newline character.

```
spc is (osp., (either nl. , osp.
               or     nil.))
```

Having defined spc, we may now modify for example the group, Boolean expression, to accept a free format input.

```
recursive Boolean expression is
          (either simple expression
           or     Boolean operator, spc, '(', spc,
                  operand list, spc, ')')
```

It is possible to have the BCL System remove all the spaces and EOR markers, but this is not always desirable; for example, spaces in string constants are of importance in most languages. Omission of EOR markers excludes the possibility of reporting error messages at the end of the input program under translation, for if these messages are to be printed at the end, they should have some reference to the program lines, but without EOR markers such a reference could not be constructed.

5. Repeated Elements

As was seen earlier, the {element}* notation in extended BNF is described by repeat (element) in BCL; the element is repeated until a failure occurs. The unit thus formed can be expressed in terms of a compound alternative. For example,

```
    repeat(',',Boolean expression)
```

is equivalent to:

```
    (either ',', Boolean expression,
            (either ',', Boolean expression),
                .
                   .
                      (either ',',Boolean expression
                       or     nil.)
                   .
                .
            or nil.)
    or nil.)
```

As can be seen from the above form, a repeat can also be regarded as a primitive unit which always returns the value true.

6. Effectiveness for Semantic Analysis

In comparison with languages like Fortran and Algol, BCL cannot be regarded as totally procedural. In order to perform semantic, as well as syntax analysis, the language should be capable of representing procedural algorithms. These are needed, for example, to search a symbol table, or to generate object code. It can be easily demonstrated that BCL lends itself to this requirement.

Five control structures, common to most procedural languages, are here expressed in terms of BCL structures. Of these, the first three are loops and the last two are conditional structures.

1) The construction while-do defined as:

   while <Boolean expression> do <block>

   can be described by:

   repeat (if <Boolean expression>,<block>)

2) The construction do-while defined as:

   do <block> while <Boolean expression>

   takes the form:

   repeat (<block>, if <Boolean expression>)

3) The construction:

   do<block-1>if <Boolean expression> then exit; <block-2>)

   can be described by:

   repeat (<block-1>, if not <Boolean expression> ,<block-2>)

4) The construction if-then defined as:

   if <Boolean expression> then <block>

   can be represented by the compound alternative:

   (either if <Boolean expression> ,<block>

   or nil.)

5) The construction if-then-else defined as:

```
if <Boolean expression> then <unconditional structure>
                        else <block>
```

is translated into:

```
(either if <Boolean expression>, <unconditional structure>
 or    <block> )
```

All the units formed in this manner always return a true value, like assignment statements, and thus the program pointer can move sequentially.

7. Representation of Data Structures

Contiguous and linked linear lists are frequently used during the compilation time, and occasionally at runtime. In this section, BCL facilities for handling such lists are investigated.

A contiguous storage pool composed of m cells, addressed from 0 to m-1, is made available to the programmer. Any integer variable may be used to access a cell; however, this should be done through defining a 'record'. For example, a dictionary record in a linked linear dictionary might be defined as:

```
record dic is (identifier name (8), :: variable identifier
               integer type,  :: type of identifier-
                              :: either integer or Boolean
               integer mode,  :: mode of identifier-
                              :: either scalar or array
               integer loc,   :: pointer to the cell holding
                              :: the value of variable at runtime
               integer nextrec :: pointer to the next record)
```

A pointer variable, say start, is needed to point to the first dictionary record. At the beginning this pointer is initialized to zero, indicating an empty dictionary. An integer variable is also required to point to the start of the storage area which could be used to keep the dictionary. This variable, here called free, could have been initialized by a group which allocates storage areas from the storage pool for different purposes.

When a statement like register x; is parsed, information concerning the name, type, and mode of the declared variable (respectively, vname, vtype, vmode) is obtained. This information should be formed into a record and appended to the previous dictionary records.

The following group performs this task:

```
append is (name(free) = vname,  ::  insert name
           type(free) = vtype,  ::  insert type
           mode(free) = vmode,  ::  insert mode
           loc(free)  = rmemory,::  allocate runtime memory
                                ::  advance runtime memory pointer
                 rmemory  =  rmemory + constant,
a)         nextrec(free) = start, ::  append the new record
b)         start = free,  :: modify start
           free = free + 1.dic :: adjust the start of the free area)
```

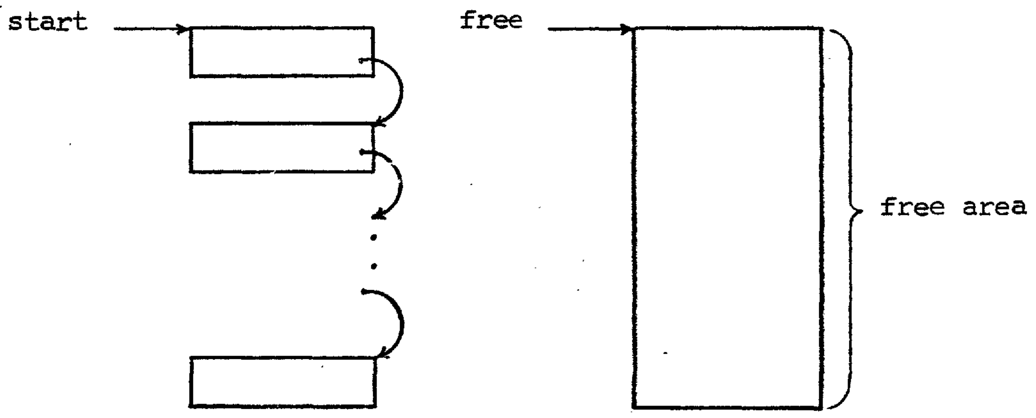


Figure 1. Storage picture before variable x is declared

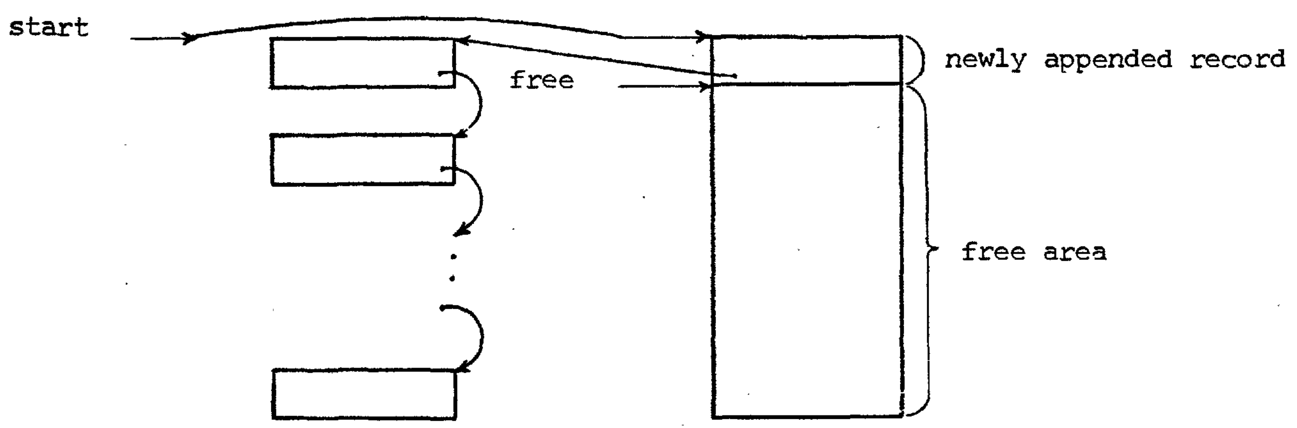


Figure 2. Storage picture after variable x is declared

The expression, 1.<record-name>, returns the number of storage cells that a record occupies; for example, 1.dic is the length of a dictionary record. Now one can easily write a group which searches the dictionary for an occurence of a certain identifier, say the one held in vname.

```
search is (ptr = start, :: initialize pointer to traverse dictionary
                repeat (if ptr ≠ 0, :: make sure that the last record
                                    :: has not yet been searched
                    if vname ≠ name(ptr), :: is the name field
                                          :: same as vname?
                   ptr = nextrec(ptr) :: if not advance the pointer))
```

After calling the group, search, any non-zero value of ptr indicates that the variable identified by vname has already been declared.

By making a storage pool available to the programmer, and by introducing the 1. notation, BCL treats linked and contiguous linear lists in the same way. For example, in order to have a sequential dictionary instead of a linked one, the only change to be made is to eliminate the field, nextrec, from the dictionary record. Consequently in the group, search, the statement, ptr = nextrec(ptr), should be replaced by ptr = ptr+1.dic and in the group, append, the statements (a) and (b) are no longer needed.

8. The Scope of Variables

The scope of a name is that region of the program throughout which that name identifies the same variable. Names declared in recursive groups have a local scope limited to the textual body of the group; those declared in non-recursive groups have a global scope covering the whole program. Similarly, the field names declared in records have a global scope; obviously such names do not correspond to a variable, and as seen in the examples they only take part in forming variables, like nextrec(free), nextrec(ptr), type (free), etc.

9. Concluding Remarks

Comments given in this section are implemention independent,with the exception of remarks on boundary alignment which refer to the current implementation under the HASP operating system on the IBM 360/65 at University College, London.

9.1 Boundary Alignment

The problem of boundary alignment in the IBM 360 series has been transmitted even to such a high level as that constituted by BCL, resulting in some inconveniences in record definition. The computing system requires that a data item 2^k Bytes long ($0 \leq k \leq 3$) should be assigned a storage area whose starting address is a multiple of the length of the data item. Therefore, the programmer should learn how many Bytes a storage cell, an integer variable, and an identifier variable occupy. It is not surprising that if the programmer does not comply with this addressing rule, he will receive a runtime message from a very low level near to the hardware, which is not of much help. The boundary alignment problem is a frequent source of error, in order to cope with which, the System should provide elaborate debugging facilities.

9.2 Runtime Errors

In general there are four causes of runtime error in BCL.

1) The programmer specifies that a certain sub-string should exist at a certain place in the input string, but this condition is not satisfied.

2) the input string has been exhausted, yet the program tries to find a match.

3) Too deep recursion.

4) Wrong addressing.

A trace routine which gives some information about the groups called and alternatives tried, since some time before the program failed, is quite helpful in detecting the first three kinds of error. The fourth kind, which becomes much more frequent as soon as list processing routines enter, is the most difficult to pinpoint. It remains undiscovered until either the boundary alignment is violated (perhaps a point in favour of boundary requirement!), or an attempt is made to write into a location outside the storage pool. Sometimes as a result of wrong addressing the source of error itself may have been destroyed, and consequently even storage dumps cannot be as helpful as they would have been otherwise.

Apart from what was mentioned above, the following point should be considered in connection with the way BCL back-tracks after a failure. The effect is that if a program failed to achieve what the programmer had in mind, he would not know how near to the result he got, unless he traced his program either by inserting acknowledgements to be output at runtime, or by using System-provided facilities. The first method expands the program, which could increase compilation and runtime errors; while the second one produces voluminous outputs if recursive and repetitive structures are traced.

9.3 Suggestions

The price paid for the universal treatment of linked and sequential lists is, as well as some notational inconvenience, that the programmer has to write his own garbage collection routines. An alternative, for example, for appending a dictionary record could be:

```
start = dic(vname,vtype,vmode,rmemory,start),
```

where the dictionary record is assumed to have been defined as before; its name, dic, acts as an operator which structures the five data items involved; the new record is then appended to the dictionary as a result of assignment. In this manner, the System administers the storage pool, rather than the programmer. Whenever it is exhausted, a garbage collection routine returns the unreferenced cells to the pool. A new type of variable, namely <u>pointer</u>, is now required. For example, the statement

```
pointer dic(start),
```

declares that start is a pointer to dictionary records (and nothing else). Thus, the discovery of errors caused by wrong addressing is not in general delayed as long as before, and the effect is not as drastic. The garbage collection routine uses pointer variables to begin collection.

Apart from pointers, inclusion of Boolean type variables and expressions is of help for safety reasons, as well as the new dimension they thus would create.

Inclusion of packing and unpacking facilities would allow efficient use of store; nevertheless, they are not of paramount importance since assembly language routines may be included in a BCL program as separate groups.

Since BCL limits itself to strictly local and global variables, inconveniences arise in connection with recursive groups. If such a group is to pass a value held in a local variable to another group, first the value must be transferred to a global variable. This difficulty can be overcome through establishing a parameter passing mechanism.

9.4 Efficiency and Effectiveness

Two remarks concerned with efficiency: firstly, writing a compiler in BCL, in contrast to an assembly language, eliminates the lexical analysis phase. If the grammar as implemented in the compiler necessitates back-tracking, the lexical analysis phase is repeated, and the user pays a price for not doing the compilation in two phases; secondly, since the alternatives in a compound are tried serially, the runtime increases drastically as the compiler grows. The combination of these two effects could result in a very poor runtime efficiency.

Because of its correspondence to the BNF notation, and its simple way of handling semantics, BCL is an easy language to use for translator writing, but not so easy to debug. Its modularity makes it helpful for language design where one needs to produce quickly a translator, and to modify it frequently. BCL manages to implement a wide range of algorithms without using goto-statements which are pointed out to be a frequent source of error [19]. The interpretation of object code, produced by the user's program, requires the use of case-statements; the current case-structure of the language is not powerful enough and does not permit multiple exits without using goto-statements.

APPENDIX C.

Definition of the Syntax

This appendix is devoted to the syntax definition of the Language introduced in Chapters II to V. The definition is presented in BNF Language Description Language.

1. Constants

```
<constant> ::= <Boolean constant>|<integer constant>
<Boolean constant> ::= <binary constant>|<octal constant>|
                       <hexadecimal constant>

<binary constant> ::= <true value>|<false value>|bin <binary list>
<true value> ::= true|on
<false value> ::= false|off
<binary list> ::= <binary digit>|(<binary digit>{,<binary digit>}*)
<binary digit> ::= 1|0

<octal constant> ::= oct <octal list>
<octal list> ::= <octal digit>|(<octal digit>{,<octal digit>}*)
<octal digit> ::= 0..7

<hexadecimal constant> ::= hexa <hexa list>
<hexa list> ::= <hexa digit>|(<hexa digit>{,<hexa digit>}*)
<hexa digit> ::= 0..f

<integer constant> ::= <constant identifier>|<decimal constant>
<constant identifier> ::= <identifier>
<decimal constant> ::= {<decimal digit>}*
<decimal digit> ::= 0..9
```

2. Variables

```
<variable> ::= <simple variable>|<subscripted variable>

<simple variable> ::= <variable identifier>

<variable identifier> ::= <identifier>

<identifier> ::= <letter>{<letter>|<decimal digit>}*

<letter> ::= a..z
```

2.1 Subscripted Variables

```
<subscripted variable> ::= <vector identifier>(<vector subscript>)|
                           <matrix identifier>(<matrix subscript>)

<vector identifier> ::= <identifier>

<vector subscript> ::= <index>|<lower index> : <upper index>

<index> ::= <subscript expression>

<lower index> ::= <subscript expression>

<upper index> ::= <subscript expression>

<subscript expression> ::= <expression>


<matrix subscript> ::= <row subscript>, <column subscript>

<row subscript> ::= <subscript expression>

<column subscript> ::= <index>|<lower index> : <upper index>|<empty>

<empty> ::=
```

3. Expressions

```
<expression> ::= <simple expression>|<Boolean expression>|
                 <delayed Boolean expression>|<integer expression>|
                 <element designator>

<simple expression> ::= <constant>|<variable>
```

3.1 Boolean expressions

```
<Boolean expression> ::= <Boolean operator> <operand list>
<Boolean operator> ::= <logical operator>|<relational operator>|
                       <shift operator>|bln

<logical operator> ::= and|or|not|nand|nor|eqv|eor
<relational operator> ::= lt|le|eq|ne|ge|gt
<shift operator> ::= rshift|lshift|rcirc|lcirc
<operand list> ::= <source vector>|(<source vector>{,<source vector>}*)
```

3.2 Delayed Boolean Expressions

```
<delayed Boolean expression> ::= <delayed Boolean operator>
                                 <operand list>
<delayed Boolean operator> ::= dand|dor|dnot|dnand|dnor|deqv|deor
```

3.3 Integer Expressions

```
<integer expression> ::= <arithmetic operator> <operand list>
<arithmetic operator> ::= plus|minus|mult|div|rem|int
```

3.4 Element Designators

```
<element designator> ::= <user-defined element>|<System-defined element>
<user-defined element> ::= <element identifier>(<input list>)
<element identifier> ::= <identifier>
<input list> ::= <expression>{, <expression>}*
```

3.4.1 System-defined Elements

```
<System-defined element> ::= <binary adding element>|<bistable>

<binary adding element> ::= add(<Boolean vector>, <Boolean vector>)|
                            sub (<Boolean vector>, <Boolean vector>)|
                            add1 <Boolean vector>|
                            sub1 <Boolean vector>

<Boolean vector> ::= <variable>


<bistable> ::= <abbreviated JK-bistable>|<JK-bistable>|
               <D-bistablc>|<RS-bistable>

<abbreviated JK-bistable> ::= ajkbs (<JK input>,<activator> <initial state>)

<JK input> ::= <general Boolean expression>

<activator> ::= <general Boolean expression>

<general Boolean expression> ::= <delayed Boolean expression>|
                                 <Boolean expression>

<initial state> ::= , <binary constant>|<empty>

<JK-bistable> ::= jkbs (<J input>, <K input>,<activator> <initial state>)

<J input> ::= <general Boolean expression>

<K input> ::= <general Boolean expression>


<D-bistable> ::= dbs (<D input>, <activator> <initial state>)

<D input> ::= <general Boolean expression>


<RS-bistable> ::= rbs (<R input>, <S input> <initial state>)

<R input> ::= <general Boolean expression>

<S input> ::= <general Boolean expression>
```

4. RT-operations

```
<source vector> ::= <expression>{:<expression>}*
<destination vector> ::= <variable>{:<variable>}*
<RT-operation> ::= <destination vector> = <source vector>;
```

5. Sequential and Parallel Networks

```
<sequential network> ::= <conditional structure>|
                         <unconditional structure>
<conditional structure> ::= <if structure>
<unconditional structure> ::= <parallel network>|<block>|
                              <for structure>|
                              <case structure>|
                              <loop structure>|
                              <miscellaneous structure>

<parallel network> ::= <RT-operation>|[{<RT-operation>}+]
<block> ::= ({<sequential network>}+)
```

6. The *if* Structure

```
<if structure> ::= <simple if>|<simple if> else <sequential network>
<simple if> ::= <if clause> <unconditional structure>
<if clause> ::= if <control condition> then
<control condition> ::= <Boolean expression>
```

7. The *for* Structure

```
<for structure> ::= <for clause> do <sequential network>
<for clause> ::= <for list>|<for list> step <index step>
<for list> ::= for <for index> = <initial value> to <final value>
<for index> ::= <integer variable>
```

```
<initial value> ::= <integer constant>

<final value> ::= <integer constant>

<index step> ::= <integer constant>
```

8. The case Structure

```
<case structure> ::= case <selector> of <case body>

<selector> ::= <integer variable>

<case body> ::= [{<case primary>}+]

<case primary> ::= <label>{,<label>}* → <sequential network>

<label> ::= <integer constant>|others
```

9. Loop Structures

```
<loop structure> ::= <do structure>|<while-do structure>|
                     <do-while structure>

<do structure> ::= do <sequential network>

<while-do structure> ::= while <control condition> do
                         <sequential network>

<do-while structure> ::= do <sequential network> while
                         <control condition>
```

10. Miscellaneous Structures

```
<miscellaneous structure> ::= <simulation structure>|
                              <I/O structure>|exit;

<simulation structure> ::= initialize;|restart;|stop;

<I/O structure> ::= <input structure>|<output structure>

<input structure> ::= read <input list>;

<input list> ::= <variable>{,<variable>}*

<output structure> ::= write <output list>;|
                       newline <number of lines>;|
                       spc <number of spaces>;
```

```
<output list> ::= <variable{,<variable>}*

<number of lines> ::= (<decimal constant>)|<empty>

<number of spaces> ::= (<decimal constant>)|<empty>
```

11. Systems

```
<system> ::= {<declaration>}*<system body>

<system body> ::= <initialization> end;|
                  start; <general network> end;|
                  <initialization> start; <general network> end;

<initialization> ::= <sequential network>

<general network> ::= {<controlled network>}+
<controlled network> ::= <control condition> <control sign>
                         <sequential network>|<sequential network>

<control sign> ::= →
```

12. Declarations

```
<declaration> ::= <constant declaration>|<vector declaration>|
                  <matrix declaration>|<sub-array declaration>|
                  <integer scalar declaration>|<delay declaration>
```

12.1 Constant Declarations

```
<constant declaration> ::= constant <constant declaration list>;

<constant declaration list> ::= <constant item>{,<constant item>}*

<constant item> ::= <constant identifier> = <integer constant>
```

12.2 Vector Declarations

```
<vector declaration> ::= <vector type> <vector declaration list>;

<vector type> ::= Boolean|register|memory|switch

<vector declaration list> ::= <vector declarative>{,vector declarative>}*
```

```
<vector declarative> ::= <scalar declarator>|<vector declarator>
<scalar declarator> ::= <scalar identifier>
<scalar identifier> ::= <identifier>

<vector declarator> ::= <vector identifier>(<dimension list>)
<vector identifier> ::= <identifier>
<dimension list> ::= <bound pair>|<vector declaration list>
<bound pair> ::= <lower bound> : <upper bound>
<lower bound> ::= <integer constant>
<upper bound> ::= <integer constant>
```

12.3 Matrix Declarations

```
<matrix declaration> ::= <matrix type> <matrix declaration list>;
<matrix type> ::= Boolean matrix|register matrix|
                  memory matrix|switch matrix
<matrix declaration list> ::= <matrix declarator>{,<matrix declarator>}*
<matrix declarator> ::= <matrix identifier>(<bound pair list>)
<bound pair list> ::= <bound pair>, <bound pair>
```

12.4 Sub-array Declarations

```
<sub-array declaration> ::= sub-array <sub-array list>;
<sub-array list> ::= <sub-array item>{,<sub-array item>}*
<sub-array item> ::= <declarative item> = <array segment>
<declarative item> ::= <scalar declarator>|<vector declarator>|
                       <matrix declarator>

<array segment> ::= <vector segment>|<matrix segment>
<vector segment> ::= <vector identifier>|<vector identifier>(<bound pair>)
<matrix segment> ::= <matrix identifier>(<row selector>,<column selector>)|
                     <matrix identifier>
```

<row selector> ::= <integer constant>

<column selector> ::= <bound pair>|<integer constant>|<empty>

12.5 Integer Scalars

<integer scalar declaration> ::= integer <scalar list>;

<scalar list> ::= <scalar identifier>{,<scalar identifier>}*

12.6 delay Declarations

<delay declaration> ::= delay <delay element list>;

<delay element list> ::= <delay element>{,<delay element>}*

<delay element> ::= <delay identifier>|<delay identifier>(<bound pair>)

<delay identifier> ::= <identifier>

13. User-defined Elements

```
<element declaration> ::= element <output declaration> =
                          (<formal input list>)
                          <element body>

<output declaration> ::= <vector type> <vector declarative>|
                         <matrix type> <matrix declarator>|
                         integer <scalar identifier>|
                         delay <delay element>

<formal input list> ::= {<declaration>}+

<element body> ::= {<declaration>}*<sequential network>
```

REFERENCES

1. Baray, M.B., Su, S.Y.H., 'A Digital System Modeling Philosophy and Design Language', Proceedings of the SHARE.ACM.IEEE Design Automation Workshop, 1971.

2. Barbacci, M.R., 'A Comparison of Register Transfer Languages for Describing Computers and Digital Systems', Department of Computer Science, Carnegie-Mellon University, 1973.

3. Barbacci, M., Bell, C.G., Newell, A., 'ISP: A Language to Describe Instruction Sets and other Register Transfer Systems', Department of Computer Science, Carnegie-Mellon University, 1973.

4. Barron, D.W., et al, 'The Main Features of CPL', Computer Journal, Vol. 6, No. 2, 1963.

5. BCPL Reference Manual, Essex University System, 1970.

6. Bell, C.G., et al, 'Register Transfer Modules (RTMs) for Understanding Digital Systems Design', IEEE Computer Conference, COMPCON 72, 1972.

7. Bell, C.G., Newell, A., 'Computer Structures: Readings and Examples', McGraw-Hill Book Company, 1971.

8. Bell, C.G., Grason, J., Newell, A., 'Designing Computers and Digital Systems Using PDP-16 Register Transfer Modules', Digital Press, 1972.

9. Bensky, L.S., 'Block Diagrams in Logic Design', Proceedings of the WJCC, 1958.

10. Bogo, G., et al, 'CASSANDRE and the Computer Aided Logical Systems Design', IFIP 71, North-Holland Publishing Company, 1972.

11. Breuer, M.A., 'Recent Developments in the Automated Design and Analysis of Digital Systems', Proceedings of the IEEE, Vol. 60, No. 1, 1972.

12. Brough, D.R., 'An Introduction to BCL', University of London, Institute of Computer Science, ICSI 452, November 1972.

13. Burnett, G.J., 'A Design Language for Digital Systems', M.Sc. Thesis, EE Department, MIT, 1965.

14. Chu, Y., 'An ALGOL - like Computer Design Language', CACM, Vol. 8, No. 10, 1965.

15. Chu, Y., 'Introducing the Computer Design Language', IEEE Computer Conference: COMPCON 72, 1972.

16. Chu, Y., 'Computer Organization and Micro-programming', Prentice-Hall, 1972.

17. Dahl, O.-J., Dijkstra, E.W., Hoare, C.A.R., 'Structured Programming', Academic Press, 1972.

18. Digital Equipment Corporation, 'Logic Handbook', 1971.

19. Dijkstra, E.W., 'Go To Statement Considered Harmful', Letter to the Editor, CACM, Vol. 11, No. 3, 1968.

20. Duley, J.R., Dietmeyer, D.L., 'A Digital System Design Language (DDL)', IEEE Transactions on Computers, Vol. 17, No. 9, 1968.

21. Ellis, R.A., Franklin, M.A., 'High-Level Logic Modules: A Qualitative Comparison', IEEE Computer Conference: COMPCON 72, 1972.

22. Falkoff, A.D., Iverson, K.E., Sussenguth, E.H., 'A Formal Description of System/360', IBM Systems Journal, Vol. 3, No. 3, 1964.

23. Flake, P.L., 'HILO - A Logic System Simulation Language', Seminar given at the Department of Electrical Engineering, Brunel University, November 1973.

24. Friedman, T.D., Yang, S., 'Methods Used in an Automatic Logic Design Generator (ALERT)', IEEE Transactions on Computers, Vol. 18, No. 7, 1969.

25. Gorman, D.F., Anderson, J.P., 'A Logic Design Translator', Proceedings of the FJCC, 1962.

26. Gould, I.H., 'Logic Design Lecture Notes', University of London, Institute of Computer Science, 1972-73.

27. Gould, I.H., 'First Thoughts on a Simulation Facility', private communication to the author, 1971.

28. Gries, D., 'Compiler Construction for Digital Computers', John Wiley & Sons, 1971.

29. Gwendolyn, G.H., 'Computer-Aided Design: Simulation of Digital Design Logic', IEEE Transactions on Computers, Vol. 18, No. 1, 1969.

30. Hesse, V.L., 'The Advantages of Logical Equation Techniques in Designing Digital Computers', Proceedings of the WJCC, 1958.

31. Hopgood, F.R.A., 'Compiling Techniques', Macdonald, 1969.

32. IBM 1130 Functional Characteristics, File No. 1130-01, Order No. GA26-5881-6, April 1972.

33. Iverson, K.E., 'A Common Language for Hardware, Software, and Applications', Proceedings of the FJCC, 1962.

34. Iverson, K.E., 'A Programming Language', Wiley, 1962.

35. Ledgard, H.F., 'The Case for Structured Programming', BIT, Vol. 14, No. 1, 1974.

36. Lewin, D., 'Theory and Design of Digital Computers', Nelson, 1972.

37. Lindsey, C.H., Van Der Meulen, S.G., 'Informal Introduction to ALGOL 68', North-Holland Publishing Company, 1971.

38. MacKinnon, A., 'Flow Charts Methods of Logic Design', Computer Design, February 1968.

39. McClure, R.M., 'A Programming Language for Simulating Digital Systems', JACM, Col. 12, No. 1, 1965.

40. Musgrave, G., White, I.J., 'Program for Digital Simulation Logic', University of Bradford, Postgraduate School of Electrical & Electronic Engineering, Report No. 90, November 1971.

41. Naur, P., Woodger, M., (Editors), 'Revised Report on the Algorithmic Language Algol 60', CACM, Vol. 6, No. 1, 1963.

42. Parnas, D.L., 'A Language for Describing the Function of Synchronous Systems', CACM, Vol. 9, No. 2, 1966.

43. PDP 11 Processor Handbook, Digital Equipment Corporation, 1971.

44. Proctor, R.M., 'A Logic Design Translator Experiment Demonstrating Relationships of Language to Systems and Logic Design', IEEE Transactions on Computers, Vol. 13, No. 8, 1964.

45. Reeves, C.M., 'An Introduction to Logical Design of Digital Circuits', Cambridge University Press, 1972.

46. Richards, R.K., 'Logic Design Methods', Proceedings of the WJCC, 1958.

47. Schlaeppi, H.P., 'A Formal Language for Describing Machine Logic, Timing, and Sequencing (LOTIS)', IEEE Transactions on Computers, Vol. 13, No. 8, 1964.

48. Schorr, H., 'Computer-Aided Digital System Design and Analysis Using a Register Transfer Language', IEEE Transactions on Computers, Vol. 13, No. 12, 1964.

49. Shahdad, B., 'Computer-Aided Teaching of Low Level Machine Architecture: Design Considerations', University of London, Institute of Computer Science, ICSI 407, May 1972.

50. Shahdad, B., 'Computer-Aided Design of Digital Systems: SEEMA, A Multi-Level Digital System Simulator', University of London, Institute of Computer Science, ICSI 427, July 1972.

51. Shahdad, B., 'Logic Design Systems', University of London, Institute of Computer Science, ICSI 500, June 1973.

52. Shaw, B., (Editor), 'Proceedings of a Seminar on the Teaching of Computer Design', University of Newcastle upon Tyne Computing Laboratory, 1972.

53. Stabler, E.P., 'System Description Languages', IEEE Transactions on Computers, Vol. 19, No. 12, 1970.

54. Whitney, G.E., Tulloss, R.E., 'The Best Language: A Language for Use in Simulation of Digital Circuits', IEEE Computer Conference: COMPCON 72, 1972.

55. Woodger, M., 'On Semantic Levels in Programming', IFIP 1971, North-Holland Publishing Company, 1972.

56. Woodger, M., 'Levels of Language', High Level Languages, State of the Art Report No. 7, Infotech Education Ltd., 1972.

57. Wirth, N., Hoare, C.A.R., 'A Contribution to the Development of ALGOL', CACM, Vol. 9, No. 6, 1966.

58. Wirth, N., 'Program Development by Stepwise Refinement', CACM, Vol. 14, No. 4, 1971.

59. Zissos, D., 'Logic Design Algorithms', Oxford University Press, 1972.