Problem orientated language for logic design

A. Kaletzky and D. W. Lewin

Department of Electrical Engineering and Electronics, Brunel University, Kingston Lane, Uxbridge, Middlesex UB8 3PH

Existing specification methods used in computer aided design of logic systems are reviewed. A new language, based on ALGOL 68 is proposed as a high level method of functional specification. The implementation of some subsets of the language is described.

(Received July 1975)

1. Introduction

In the design of large digital systems, CAD techniques have proved to be extremely effective. This is particularly so in the production engineering phases of design where logistics and documentation have been generated automatically, for example the layout and routing of printed circuit boards, backplane wiring, etc. However, very little work has been done in the general area of logic circuit synthesis due mainly to the lack of a suitable method of specifying many variable systems. One logic design system which has been described in the literature is the CALD (Lewin, Purslow and Bennetts, 1972) system which handles combinational and sequential circuits with up to 20 variables, using batch processing techniques. A major disadvantage of this system is that the synthesis routines require a tabular input to be specified by the designer which severely limits the size of problem which can be conveniently handled. One solution to this problem is to develop an interactive logic design language which allows the designer to describe a subsystem component from a standard teletype terminal. The language should allow a partial specification of the logic circuit to be declared (that is, only the pertinent input/output conditions are required to be specified) with the computer software continually assembling and checking the final tabular description (or its equivalent). For example in the case of combinational circuits, the designer would first declare the number and type of input and output variables and then carry on to specify the outputs required for a particular combination of some subset of input variables. This is analogous to the usual method of logic design where outputs are specified only for combinations of input variables of specific interest.

In addition the designer should also be able to declare a functional description of the circuit of the form $x = f(y)$ where x and y are input and output vectors and f a functional relationship which may be either a primitive or derived function. Facilities must also exist for the specification of 'don't care' terms. Another requirement is that the language statements must be checked for logical errors and conflicts, in addition to the usual syntax checking procedures. A conflict can arise in a partial specification due to an output being simultaneously required to be logical 0 or 1 or a 'don't care' condition.

To achieve the necessary interaction the designer must be constantly updated with computer adjusted messages informing him of inconsistencies, errors and when further clarification is needed.

In the system to be described the logical function of the circuit is specified by a procedural description in terms of elementary combinational operations. Emphasis is placed on the use of 'don't care' terms and on the treatment of logically conflicting definitions. In the case of sequential logic, the designer specifies the control algorithm as a conventional flowchart, which is then described using the design language and automatically translated into a state table description. A translator to the HILO logic simulation language is also being developed (Flake, Musgrave and Shorland, 1975; Flake and Musgrave, 1975).

HILO is a structural hierarchical hardware specification language capable of system simulation and fault insertion. The language is approximately equivalent to a macro assembler, the basic data objects being signal wires and bistables. Basic functions are elementary logic elements—AND, OR, etc. More complex functions are defined in HILO by structures similar to FORTRAN subroutines and out-of-line functions. An extension allows the behavioural specification of finite state machines using a FORTRAN-like logical IF and GOTO constructions and labels.

In considering the language the basic approach has been to reject the purely behavioural, black box model of classical automata theory in favour of a functional or pseudo-structural concept. The value of this approach has been eloquently argued by Dijkstra (Dahl, Dijkstra and Hoare, 1972). We regard his arguments as no less applicable to logic design than to programming.

Finally, although it has been argued that linguistic representations are less amenable to mathematical treatment than graphical ones, this does not appear to be inherent. Indeed, if the linguistic specification is rigidly constrained by the rules of structured programming it gives rise to a structure far simpler than, for example a Petri-net. (Petri, 1962).

Bearing these principles in mind a logic design language has been proposed, based on ALGOL 68 syntax, which can act as the input medium for the CALD system. The language has been formally defined, and is described together with the examples of its use, in the following sections of the paper.

2. Language requirements

2.1. *General*

The general requirements for CAD systems imply that the specification procedures used must be as far as possible similar to existing design methods. Input procedures should be interactive, and require the minimum of programming skill for their application. Automatic checking for errors and conflicts should be an integral part of the syntax checking routines. The CAD system (in this case CALD) must be accessible through a remote teletype or graphics terminal on a timesharing system. Any software developed should be easily portable, allowing the system to be operated on different computer installations.

Input procedures for CAD systems consist normally of two parts: an overall command structure which enables the user to interact with the program, and a problem oriented language used to specify the system being designed. In view of the facilities available in modern time sharing operating systems, which enable many of the overall command structure requirements to be implemented directly, the authors decided to concentrate initially on the requirements of the problem oriented language.

The justification for this is that with modern editing techniques (e.g. GEORGE 3 editor) the language can be a pseudo-batch one in the first instance, with a large amount of interaction being provided by the user editing source files. Initially, the source file can be set up by using an interactive macrogenerator in which the actual parameters, rather than being specified after each macro call, are input from the teletype when they are required in the text.

Furthermore, as the system must be easily portable, and as any user will have to be familiar with the operating system of the host machine, it is proposed to use the host machine's editor. The macrogenerator should be a part of the CALD system, but its development will be delayed until the target language is ready.

The CALD synthesis modules require tabular input in the form of ON and OFF vertex arrays and state tables. Thus the language, even if it is to be based on a functional approach should be capable of specifying subsystem components using Boolean expressions. Facilities must also be available for the explicit declaration of small truth tables. However, it is intended that the language should be used primarily for the design of large circuits. It is obvious that the use of a complete tabular specification for large variable circuits is out of the question, due to the excessive storage requirements.

2.2. *Circuit model*

The model used in the specification language considers a combinational circuit to consist of 'wires' carrying 'signals' which can be 0, 1 or X (don't care). These signals (shown in **Fig. 1**) may be 'rowed' (to form a highway) so that a vector of wires representing, e.g. a binary integer, can be treated as one entity. Unlimited fan out is allowed from any signal wire.

The wires form the interconnection paths between combinational logic elements as in the usual logic circuit network. Each element in the network has a set of input wires, on which signals must be present before it can operate and a set of output wires, on which it creates signals. The sequence of operation is shown in **Fig. 2**, the outputs appearing *after* the input wires are activated. The logic elements may be intrinsically defined in the problem orientated language (using basic boolean connectives such as $\vee$ and $\wedge$, etc.) or may be other combinational units, defined by the designer elsewhere.

The operation of a model proceeds on a step by step basis (corresponding to the 'elaboration' of its language description) and is as follows:

1. Initially all wires are in the undefined state, with no signals on them.
2. Signals appear on all input wires simultaneously.

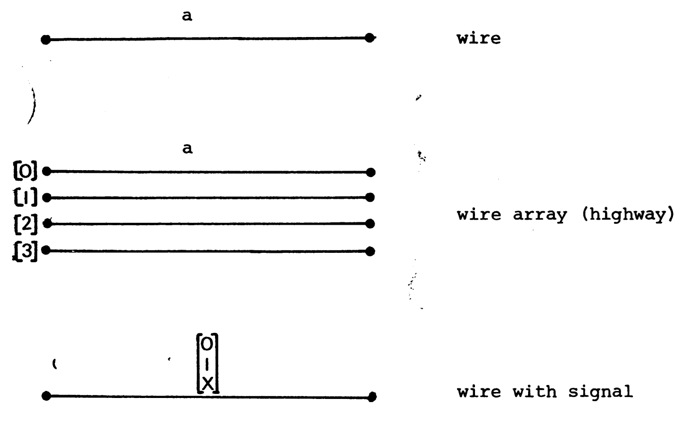


Fig. 1 Signal/wire convention

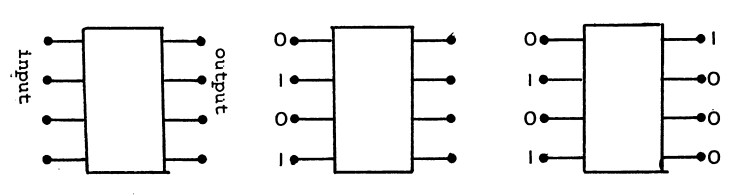
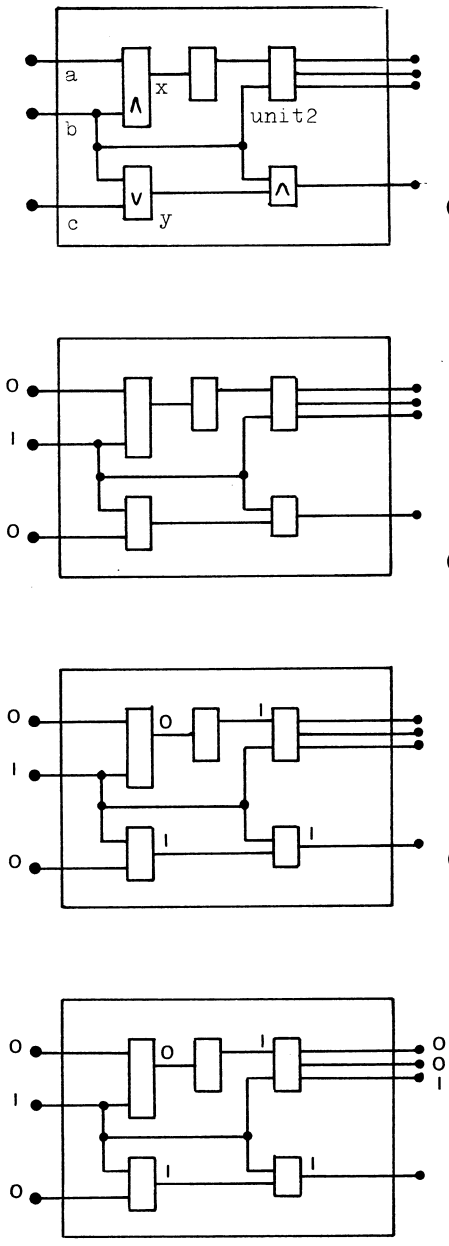


Fig. 2 Operation of logic element



(*a*) **Original circuit— no signals defined**

(*b*) **Input signals defined**

(*c*) **Signals progress through network**

(*d*) **Final outputs generated**

Fig. 3 Operation of subsystem model

3. All elements which have signals on each input wire and no signal on any output wire are 'operated', i.e. create signals on their output wires.
4. Step 3 is repeated until
5. All output wires have signals on them.

During this process (illustrated in **Fig. 3**) all signals including the input ones, must persist on the wires on which they were created. Note that the model can only represent combinational circuits since feedback loops are prohibited by the condition that all wires are initially undefined.

For example, Fig. 3 can be described by:

```
proc unit1 = (signal a, b, c) [1:4] signal
begin signal x = a ∧ b,
      signal y = b ∨ c,
      [1:3] signal w = unit2(−x, b),
      signal u = y ∧ b;
      signal z = (w[1], w[2], w[3], u);
      z comment this is the one statement allowed which
                delivers the value of the result
        comment
end
```

Two approaches, not mutually exclusive, are possible for extending the model to cover sequential circuits. The first is to allow modes such as **ref signal** (a one bit store) and **ref** [1:*n*] **signal** (a vector of *n* bits). This makes a sequential specification similar to a conventional register transfer language. Parallel operations can be specified as ALGOL 68 collateral clauses and an algorithm for the extraction of control states exists (Friedman and Yang, 1969).

Work is also being carried out in parallel on the specification and evaluation of hierarchical (that is structured) complex logic systems (Foo and Musgrave, 1975) and the language used in this paper can be used for specifying the initial input procedures.

When specifying sequential circuits the basic storage requirements are satisfied by using the ALGOL 68 assignation. For example, in the assignation *a*:= *b*; *a* is of mode **ref signal** and is interpreted as the input to a bistable, whereas *b* can be either of mode signal, in which case it is simply a wire, or if it is **ref signal**; it is automatically 'dereferenced' to signal by the rules of ALGOL 68. The dereferencing is interpreted to mean 'the wire with the output of bistable *b*'. From an input of this type it is possible to generate a state table for the control structure together with the corresponding data structure, as an input to the CALD system.

The second approach is far more restricted, but has the advantage of requiring a far simpler language subset and corresponding to a well known manual design method—the Algorithmic State Machine concept, described by Clare (1973). The main simplification is that the data path (i.e. internal storage and any combinational operators used) is excluded from the sequential control unit being designed. Thus, there is no need to introduce **ref signal** for internal variables—the state variables are implicit in the position of the language element being elaborated. This model owes its 'sequentionality' to **goto** statements and labels—it corresponds almost exactly to the programming flowchart. It is in fact Clare's class 4 machine.

The ASM approach has been selected for development initially. This does not rule out the eventual implementation of an integrated logic specification language, but at this stage it is more advisable to separate the problems of control and data path specification.

2.3. *Language*

Before a model of this sort can be processed, it must be described to the computer in a suitable format. It was thought advantageous to base the problem orientated language on the syntax of a well established language structure, which alleviates the design problem and increases the acceptance of the language among users. **Table 1** compares the distinctive features of current high level languages which could be employed for this purpose with what is available in low level languages.

Of the languages considered, PL/1 (Bates and Douglas, 1970) was thought to have a syntactic structure which would be time consuming and difficult to implement. Iverson (1972) (APL) uses a peculiar symbol set, has some unfamiliar features (e.g. evaluation of expressions from right to left), and tends to be too concise—it is difficult to see (without some considerable experience) what an Iverson program is intended to do.

ALGOL 68 (van Wijngaarden, Mailloux, Peck and Koster, 1970; Lindsey and van der Menlen, 1971) has several desirable features—the distinction between reference to a value and the value itself, the prelude in which operators may be defined, and the capability of automatically selecting the version of the operator suitable for the operands. In addition there is a very powerful and convenient set of conditional clauses.

Using ALGOL 68 syntax the combinational units in the model could be packaged as operators or procedures. The model requires after a unit is elaborated, that its internal wires go into an undefined state till the next elaboration. This is analogous to what happens in ALGOL 68 (and 60) block structure with locally generated objects.

Table 1 Comparison of high and low level languages

High level language	Low level language
Semantics depend on context	Semantics do not depend on context
High object source ratios	Low object source ratios
Possibility of statement reordering and optimisation	No reordering or optimisation
Library routines including implicitly	Explicit calls to library routines
Operators context sensitive compile differently with different operands	Instruction always compiles the same op code
Many errors detected as syntax errors	Few errors detected as syntax errors

3. Description of language

3.1. *Combinational networks*

As discussed earlier, the syntax of the language is based on a small subset of ALGOL 68. This enables Backus–Naur form, rather than the more elaborate two phase grammar expansion to be used for its formal definition, as shown in the Appendix. The definition of a combinational unit is packaged as an ALGOL 68 routine, that is an operator or procedure denotation. The routine consists of a header, specifying the connections between the unit and its environment, (i.e. the interrelationships with other modules at the same system level) the names and modes of input parameters, output mode, the name of the routine, etc. The header is followed by the body of the routine, which corresponds to the wires and operators inside the model.

As the routine represents a combinational operator, only objects which have the mode **signal** (i.e. signal values which are attached to a name for the entire duration of the routine's elaboration) are permitted. The mode **ref signal**, which would represent a store of one bit is forbidden.

Objects of the usual ALGOL 68 modes **bool, int** and references to them may be used in a restricted sense, to make the specification more concise. They do not, however, have any physical interpretation in the model—thus they cannot be used as parameters or delivered results.

Objects of mode **signal** consist of an identifier associated with a value of 0, 1 or *X* (don't care) or simply a literal value of 0, 1 or *X*. The former case represents a wire with a signal on it. A constant signal may be associated with a wire by a declaration, e.g. **signal** *awire* = 1. Note that wire *awire* will have a 1 signal on it in every elaboration. An input signal may also come in as a parameter; for example:

proc aproc = (**signal** *awire*, *bwire*) **signal**:

where wire *awire* is now the wire from the first input port. Note that parameters are passed by value rather than by name or reference since other mechanisms are not possible because of the prohibition of **ref signal**. Alternatively *awire* may be the output wire of an internal operator, such as:

signal *awire* = *bwire* + *cwire*
signal *awire* = *add* (*bwire*, *cwire*)

In the first case, *awire* is the output resulting from an instance of the operator associated with the token + and the operands *bwire* and *cwire*. In the second case, *awire* is the output resulting from an instance of the unit associated with the procedure called '*add*' and having inputs *bwire* and *cwire*.

It should be stressed that these declarations are not store assignments which in some languages are also denoted by the token '='. In ALGOL 68 assignations are denoted by ':=' and are forbidden in the case of a combinational specification with objects of mode **signal**. This is because any assignation which has mode **signal** on its right hand side must also have mode **ref signal** on its left hand side; this however would represent a store and is therefore forbidden.

The declarations as described above are sufficient to specify a combinational model. However, such a description would not be elegant, and further facilities have been introduced to facilitate circuit specification. For example, a vector (or row) of wires may be represented by one identifier as:

[1:3] **signal** *arow* = (*awire*, *bwire*, *cwire*)

this being equivalent to:

[1:3] **signal** *arow* = ; *arow* [1] = *awire*, *arow* [2] = *bwire*, *arow* [3] = *cwire* .

Note that brackets are used in the conventional manner to indicate subscripting when row elements are used, and also for declaring the dimensions of a row (in certain circumstances the dimensions may initially be left undeclared).

Modes such as **ref** [] **signal**, **ref** [][] **signal**, **ref** [1:3] **signal**, etc. are forbidden as they would represent storage.

Declarations of the form [*a*:*b*] **signal** *arow* = ; are the only non-ALGOL 68 syntactic construction in the language. They serve to warn the translator that an object *arow* of mode [1:3] **signal** will be defined by declaring its individual elements. (Note that [] **signal** and **signal** are different modes.)

Facilities also exist in the Language for defining complex operators such as exclusive or, comparator etc. in terms of elementary operators or as primitives (if their hardware implementation already exists). The constructions used are

priority ⟨operator token⟩ = ⟨integer⟩

to define priority for syntax analysis and

op ⟨optoken⟩ = ⟨definition of logic circuit as with routine⟩

The second part of Example 2 illustrates the use of the exclusive or operator.

An additional feature of ALGOL 68 is that the same operator token (e.g. +) can represent many different operators, the actual operator selected being dependent on the modes of the operands. Thus

[1:24] **signal** *a*, *b*
[1:24] **signal** *c* = *a* + *b*

will select a 24 bit parallel adder but

[1:4] **signal** *a*, *b*
[1:4] **signal** *c* = *a* + *b*

will select a four bit chip adder.

This feature is useful in descriptions of complex circuits containing separately designed modules such as LSI chips.

After all the relevant signals have been declared, the next stage is the elaboration of a statement which delivers the result of the routine. The result may be of mode **signal**, [] **signal**, [][] **signal**, etc. Note that it is impossible to have several alternative statements in one routine. Loops are prohibited as they can make the unit sequential.

3.2. *Sequential circuit*

We have chosen to base the sequential subset for initial implementation on Clare's ASM (Clare, 1973). Thus, the only features necessary are a way of describing a state and a means of selecting the next state. There is no necessity for explicit variable stores—the only internal variable is implicit, the current state.

Thus, the typical construction of a sequential specification is a conditional:

label: **if** *condition* **then** *out* 1,/*out* 2, *out* 3, **goto** *label* 1
else *out* 4, **goto** *label* 2
fi

The label serves only as a target for a **goto** selecting the next state. The condition is a boolean expression made up to input variables. The outs are output signals—technically, invocations of procedures. Constructs separated by commas may be performed in parallel, while those separated by semicolons are in sequence.

4. Implementation

At this stage, it was considered more important to study the particular problems of logic design with text input rather than engage in the development of sophisticated software with complex syntax analysis, etc. To reduce the software effort, small subsets of the language were selected for initial implementation. The aim is to deal with the specific problems of this project while avoiding, for the time being, the general problems of implementing a large and rich language.

These initial implementations used as research vehicles, will indicate empirically which language features are of most use in practical designs.

4.1. *Combinational circuit*

Input variables are declared in the header as described for the full language. Output variables are declared as a single array (Lindsey and van der Meulen, 1971) and indexed through to 20.

The basic construct selected is shown below:

if *condition* **then** *signal declaration*
else *signal declaration*
fi

Only output signals may be declared in the subset—internal signals are prohibited.

The condition, if it is not a simple Boolean product, is transformed into a sum-of-products form. The whole construct is then repeated with each product as a condition.

ON and OFF terms are stated as lists for each output variable. An

if *simple product condition* **then** *signal declaration* **fi**

construct can be added to the relevant list (or if the signal is set to 1, off if zero, both if don't care). However, the condition may not include every input variable as a literal. Thus, either the condition must be expanded to a large number of 'all-literal' conditions, or some reduced specification found. An obvious solution is to use a data representation consisting of two words per item above the bits in the first word indicate the presence or absence of an input literal, and the second word their values.

Conflict checking is facilitated by this representation. The criterion for conflict freedom is that each new item added to one list must have at least one input literal in common with each item existing on the opposite list, and the value of that literal must be different.

Although this criterion is expensive to evaluate in terms of computation time, if the checking is done as the specification is interactively input, the delay caused by checking will be small compared to the general overheads of input.

4.2. *Sequential circuit*

The aim in this case was to translate the specification to the HILO language (Flake and Musgrave, 1975) rather than directly to a state table. In fact, the chosen method is to translate to an intermediate language, from which HILO code can be derived very easily. The intermediate language is also to be capable of being easily transformed to a state table.

The intermediate language COLT is basically the specification language with nesting removed. Nested **if** conditions are removed by working out the complete condition on which an output depends, and then reordering the conditions and outputs so that the conditions most difficult to satisfy appear first. Thus

l: **if** *a* **then if** *b* **then** *u* **else** *w* **fi**
else if *c* **then** *x* **else** *y* **fi**

```
 m: comment next state description comment
is transformed to
 l  IF(a ∧ b)OUT(u)GO(m)IF(a)OUT(w)GO(m)
    IF(c)OUT(x)GO(m)OUT(y)GO(m)
 m
```

4.3. *General*

To satisfy the portability requirement the implementation language for all software will be BCPL (Richards, 1969), which is usable on many different machines and which is itself highly portable.

As the user will have to be reasonably familiar with the operating system of the host computer, it is intended to use as much as possible of the available software. The overall command structure to control CALD will be, e.g. in the case of the ICL 1900 series machines, GEORGE 3 macros. A sufficiently comprehensive set of macros can be defined to make special control software unnecessary. Initially, the host computer's editor will be used to provide interaction with the specification source file. If experience shows this to be inadequate, it should be easy to specify and write a special purpose editor.

5. Conclusions

A problem orientated procedure language for the specification and design of combinational and sequential circuits has been proposed. Initial work suggests that this language is a convenient means of generating logic specifications using a practical model easily assimilated by the designer. It has also been shown that ALGOL 68 syntax provides a convenient basis for the logic specification language.

Given the present level of technology, the partitioning of the system being designed has to be done by the designer rather than automatically by the CAD system. This does not limit the designer in any way, since this is the natural method of designing complex systems. However, it does mean that system optimisation is restricted to subsystem level. Currently work is in progress on implementing the language subset, using BCPL as the programming medium on the ICL 1900 range under GEORGE 3 and on the PDP 11 under DOS.

6. Examples

This section illustrates some typical problems specified in the design language. Example 1 is a functional representation of a combinational circuit for code conversion, while example 2 shows the explicit declaration of a truth table. The specification of a simple flowchart (shown in **Fig. 4**) is illustrated in example 3, with a more complex example, that of Clare's blackjack machine (1973), in example 4.

Example 1

```
comment this procedure converts four-bit BCD to excess-3 code
  comment
proc excess3 = ([1:4] signal binary) [1:4] signal;
begin if abs(binary) 10 then [1:2] signal three = (1, 1);
                             binary + three
                             else (X, X, X, X)
     fi
end
```

Example 2

```
comment this procedure converts from pure binary to Gray code;
  the method of specifying a fully defined truth-table is illustrated
  comment
proc gray = ([1:4] signal i) [1:4] signal:
begin if i = (0, 0, 0, 0) then (0, 0, 0, 0)
      elsf i = (0, 0, 0, 1) then (0, 0, 0, 1)
      elsf i = (0, 0, 1, 0) then (0, 0, 1, 1)
      elsf i = (0, 0, 1, 1) then (0, 0, 1, 0)
      elsf i = (0, 1, 0, 0) then (0, 1, 1, 0)
```

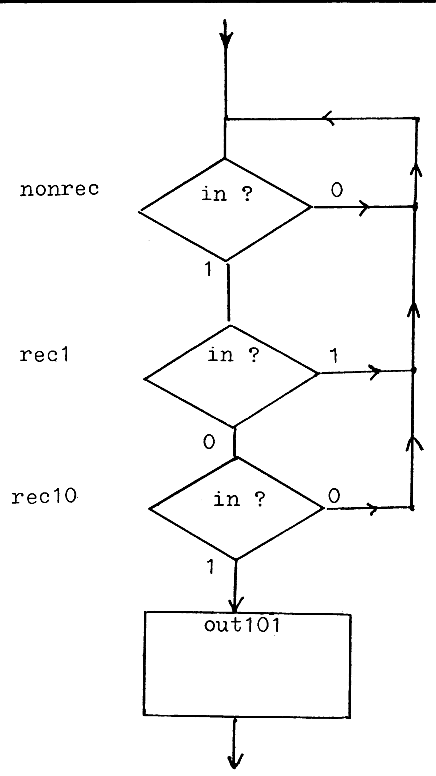


Fig. 4 Flowcharts for example 3

```
      elsf i = (0, 1, 0, 1) then (0, 1, 1, 1)
      elsf i = (0, 1, 1, 0) then (0, 1, 0, 1)
      elsf i = (0, 1, 1, 1) then (0, 1, 0, 0)
      elsf i = (1, 0, 0, 0) then (1, 1, 0, 0)
      elsf i = (1, 0, 0, 1) then (1, 1, 0, 1)
      elsf i = (1, 0, 1, 0) then (1, 1, 1, 1)
      elsf i = (1, 0, 1, 1) then (1, 1, 1, 0)
      elsf i = (1, 1, 0, 0) then (1, 0, 1, 0)
      elsf i = (1, 1, 0, 0) then (1, 0, 1, 0)
      elsf i = (1, 1, 0, 1) then (1, 0, 1, 1)
      elsf i = (1, 1, 1, 0) then (1, 0, 0, 1)
      elsf i = (1, 1, 1, 1) then (1, 0, 0, 0)
end
comment a far more elegant way of achieving the same result is
  comment
op ⊕ = ([x, y] signal a, b) [x, y] signal:
begin [x, y] signal z =,
for i from x to y do
  z[i] = (a[i] ∨ b[i]) ∧ −(a[i] ∧ b[i]);
  z
end
priority ⊕ = 4
proc gray 1 = ([1:4] signal i) [1:4] signal:
begin [1:4] signal j = (0, i[1], i[2], i[3]);
     i ⊕ j
end
```

Example 3 (sequential)

```
comment this unit recognises a 101 sequence in a one bit input
  stream comment
proc rec101 = (ref proc out101, sigin in) void:
begin
nonrec: if in then goto rec1 else goto nonrec fi;
rec1:   if −in then goto rec10 else goto nonrec fi;
rec10:  if in then out101 else goto nonrec fi
end
```

Example 4

comment *this is a description of Clare's blackjack machine*

```
comment
proc blackjack = (ref proc HTVC, HADD, HT10, HJ50,
                           HT22, HK50, HCLS, IHHIT,
                           IHBRK, IHSTND,
                  sigin YCRD, YF50, YACE, YG16, YG21)
begin
b: HTVC;
a: skip, if YCRD then goto a fi;
c: HADD, if –YF50 ∨ YACE
            then HT10, HJ50, goto c
            fi;
d: IHHIT,
   if YG16
   then if YG21  then if YF50
                      then HT22, HK50, goto c
                      else goto g
                      fi
                 else goto h
        fi
   else if YCRD then goto b
                else goto d
        fi
   fi;
g: IHBRK, HCLS, HK50
   if YCRD then goto b else goto d fi;
h: IHSTND, HCLS, HK50,
   if YCRD then goto b else goto d fi
end
```

Acknowledgement
This work has been performed under the auspices of an SRC research grant and one of the authors (A. Kaletzky) would like to acknowledge SRC for their support in providing a research studentship.

Appendix 1 Syntax of combinational language

⟨signal value⟩ ::= |0|1|x|ε|

x represents a don't care output, ε an undefined (not yet defined) signal and can be represented by a blank in the language.

⟨identifier⟩ ::= |⟨letter⟩|⟨identifier⟩⟨letter⟩|⟨identifier⟩ ⟨digit⟩|

⟨signal primary⟩ ::= |⟨signal value⟩|⟨identifier⟩|(⟨signal expression⟩)|⟨identifier⟩[⟨integer expressions⟩]|⟨closed clause delivering mode **signal**⟩|

all the identifiers above should refer to objects of mode signal and [] signal respectively.

⟨sig value set⟩ ::= |⟨signal primary⟩|⟨sig value set⟩, ⟨signal primary⟩|

⟨sig value row⟩ ::= (⟨sig value set⟩)

The above construction is used to declare a row of signals

⟨rower⟩ ::= |[]|[⟨int value⟩:⟨int value⟩]| [⟨int value⟩:⟨int value⟩ **at** ⟨int value⟩]|

⟨row primary⟩ ::= |⟨sig value row⟩|⟨signal row expression⟩| ⟨closed clause delivering [] **signal**⟩|

⟨signal multiple declaration⟩ ::= |⟨rower⟩⟨identifier⟩ = |⟨rower⟩⟨identifier⟩ = ⟨row primary⟩)

Constructions involving int or bool objects and not signal are defined as in ALGOL 68. Structures and unions are not permitted.

⟨sigmode⟩ ::= |signal|⟨rower⟩⟨signmode⟩|

⟨redeclaration⟩ ::= |[⟨int value⟩]⟨identifier⟩ = ⟨signal primary⟩

Used to define signals on a row which is declared but not defined.

⟨sig mode declaration⟩ = |⟨signal declaration⟩|⟨signal multiple declaration⟩|

⟨combden⟩ ::= ⟨combop⟩|⟨combproc⟩

A combinational unit is packaged as an operator or a routine.

⟨combop⟩ ::= |**Priority** ⟨opsymbol⟩ = ⟨digit⟩ **op** ⟨opsymbol⟩(⟨sigmode⟩, ⟨sigmode⟩) ⟨sigmode⟩:⟨combody⟩ | **op** ⟨opsymbol⟩ = (⟨sigmode⟩)⟨sigmode⟩:⟨combody⟩

An operator is either dyadic with priority or monadic with a priority greater than dyadic operators and the same for all monadic operators. All parameters must be signals, so must the result.

⟨combproc⟩ ::= **proc** ⟨identifier⟩ = (combparmpack) sigmode:combody

⟨combparmpack⟩ ::= |⟨sigmode⟩|⟨combparmpack⟩, ⟨sigmode⟩|

⟨combody⟩ ::= **begin** ⟨decprelude⟩; ⟨result interlude⟩ **end**

⟨decprelude⟩ ::= |⟨sigmode declaration⟩| ⟨interbool declaration⟩| ⟨decprelude⟩, ⟨sigmode declaration⟩| ⟨decprelude⟩, ⟨interbool declaration⟩| ⟨decprelude⟩; ⟨redeclaration⟩| ⟨decprelude⟩; ⟨decprelude⟩| ⟨decprelude⟩; ⟨decprelude⟩, ⟨redeclaration⟩

⟨statement⟩ ::= |⟨intorbool statement⟩| **goto** ⟨label⟩| **if** ⟨bool primary⟩ **then** ⟨statement⟩ **else** ⟨statement⟩ **fi**| **case** ⟨int primary⟩ **in** ⟨statement sequence⟩ **out** ⟨statement⟩ **esac**| ⟨sigmode expression⟩| ⟨intorbool expression⟩|⟨label:statement⟩|

⟨clause delivering sigmode⟩ ::= |⟨statement delivering sigmode⟩|⟨statement⟩ statement delivering signal

The object delivered by a statement is the object evaluated last during the elaboration of the statement.

⟨result sigmode⟩ ::= |⟨clause delivering sigmode⟩| ⟨label: result statement⟩|

⟨result interlude⟩ ::= |⟨result statement⟩|⟨result interlude⟩ **exit** ⟨label⟩:⟨result statement⟩|

Appendix 2 Standard prelude operators

Operator	*Operand* 1	*Operand* 2	*Result*
= ≠	signal or [ ] signal or bool or int	signal or [ ] signal or bool or int	bool (truth value of comparison)
+ – * ∧ ∨	[ ]signal	[ ]signal	[ ]signal (arithmetic function)
abs	[]signal		int (binary value)
≃	signal		signal (negation)
Other operators for int and bool as in standard ALGOL 68			

References
Bates, F. and Douglas, M. L. (1970). *Programming Language One*, Prentice-Hall, Englewood Cliffs.
Clare, C. R. (1973). *Designing Logic Systems Using State Machines*, McGraw-Hill, New York.
Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured Programming*, Academic Press, New York.

Flake, P. L., Musgrave, G., and Shorland, M. (1975). The HILO Logic Simulation Language, *Proceedings Workshop on Computer Hardware Description Languages and Their Applications*, New York.

Flake, P. L. and Musgrave, G. (1975). A Digital System Simulator—HILO, *Digital Processes*, Vol. 1, pp. 39-53.

Foo, S. Y. and Musgrave, G. (1975). Comparison of Graph Models for Computation and their Extension, *Proceedings Workshop on Computer Hardware Description Languages and their Applications*, New York.

Friedman, T. D. and Yang, S. C. (1969). Methods used in Automatic Logic Design Generator (ALERT), *IEEE Trans. on Computers*, Vol. C18, No. 7, pp. 593-614.

Iverson, K. E. (1972). *A Programming Language*, John Wiley, New York.

Lewin, D. W., Purslow, E. J., and Bennetts, R. G. (1972). Computer assisted Logic Design—The CALD System, IEE Conference on CAD, *IEE Conf. Pub.* Vol. 86, pp. 343-351.

Lindsey, C. H. and Van der Meulen, S. G. (1971). *Informal Introduction to Algol 68*, North Holland Publishing, Amsterdam.

Petri, C. A. (1962). Kommunikation mit Automaten, Ph.D. Thesis, University of Bonn. English translation: Communication with Automata Supplement to RADC-TR-65-377, Vol. 1, US Air Force, Griffiss AFB, New York, 1966.

Richards, M. (1969). BCPL—A Tool for Compiler Writing and System Programming, *AFIPS Proc.*, Vol. 34, pp. 557-566.

Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., and Koster, C. H. A. (1970). Report on the Algorithmic Language Algol 68, *Numerische Mathematik*, Vol. 14, pp. 80-218.

Book reviews

Computers in Neurobiology and Behaviour, by B. Soucek and A. D. Carlson, 1976; 324 pages. (*John Wiley*, £13·85)

Computer Technology in Neuroscience, by P. B. Brown, 1976; 650 pages. (*John Wiley: Halsted Press*, £17·75)

These two books are aimed at widely differing audiences. Consequently it is perhaps most useful to treat them individually before comprising their relative qualities.

Branko Soucek and Albert Carlson have considerable experience in writing books about computing. This book, as its name implies, is designed to provide the reader with an insight into the application of computers to the life sciences. The problem is that it is extremely difficult to provide useful information about data acquisition, computer programming, signal analysis, and simulation, in the space of less than two hundred pages. For someone fresh to the field, chapter one, although rather short, contains some interesting ideas on the mechanisms underlying neural processes. Unfortunately, the next two chapters switch from biology to the logical operations underlying the working of a computer rather rapidly. It is also difficult to believe that anyone trying to learn BASIC could get very far with the aid of this book. Nevertheless, the mere fact that the authors have included something on correlation and spectral analysis might allow biologists unfamiliar with these techniques to gain some idea of their importance.

Computer Technology in Neuroscience edited by Paul Brown is aimed at a completely different audience, namely research workers in the general area of neuroscience. A good deal of the material is, however, fairly basic. For example the chapters by A. S. French, while only providing an introduction to the analysis of neural spike trains, do provide some very useful references and have the distinct advantage of having been written by one of the leading people in the field. One of the main disadvantages of the book is that there are too many contributors which makes it rather long; consequently one had the feeling whilst reading it that you were always likely to pick up some useful ideas but a proper explanation of the work described could only be obtained by reading the original papers. Another difficulty is that the book, because of its origin in a symposium, comprises presumably the extended versions of the papers originally presented, it therefore lacks any real structure. The editor has for example, made no attempt to group the chapters in any definite way. In fact, there is loose grouping within the text, but division of papers into proper subsections with the appropriate editorial introduction would have helped enormously.

In summary both books have their good and bad points. Soucek and Carlson's book is well laid out and is therefore readable. Its disadvantages are that it tries, perhaps, to cover too much, too quickly, and jumps too rapidly from biology to computer technology. Paul Brown's book contains a great deal of valuable information for research workers, the main difficulty being that it often takes rather a long time to find it.

R. I. Kitney (London)

Programs, Machines and Computation, by K. L. Clark and D. F. Cowell, 1976; 176 pages. (*McGraw-Hill*, £5·25)

Programs and Machines, An Introduction to the Theory of Computation, by Richard Bird, 1976. (*John Wiley*, £7·90)

As their titles suggest, these two books cover the same ground: indeed, since all the authors acknowledge Scott (1967) as their inspiration, their similarity is even less surprising.

Each book covers adequately the definitions of machines and the establishment of the correctness and equivalence of programs. Both are intended primarily for computer science undergraduates and have been field tested in the authors' own teaching establishments. The authors assume, very reasonably, that their readers will have a minimal pre-knowledge and a willingness to learn. Any reader who is unfamiliar with 'modern mathematical notation', and that includes most of us over 30, will find that these books are not too easy to read. They need to be studied carefully, which is not a bad property of a book intended for undergraduates. Having read both of them within a short time I am unable to decide if my clearer understanding of the one is simply due to my having read the other first. Both books are well presented and substantial; neither is too expensive by today's standards.

On the whole I feel that either of these books represents a good buy for a student involved in the second and third year studies of most good computer science courses; there are more exercises in Bird but the Clark and Cowell is cheaper. I shall recommend to my students (and colleagues) that they invest in Clark and Cowell, mainly because of the price and a common affinity for the thinking of Dijkstra *et al.*

I leave the denotation of the (not many) typographical errors as an exercise for the (I hope many) readers.

Alan Chantler (Yelvertoft)

Reference

Scott, D. (1967). Some definitional suggestions for automata theory, *J. Comp. Sys. Sci.*, Vol. 1, pp. 187-212.