A computer model for instructional purposes

M. H. Williams and H. L. Ossher

Department of Computer Science, Rhodes University, P.O. Box 94, Grahamstown, South Africa

This paper describes a computer simulator which is used as an aid to teaching certain aspects of computer science. The properties of the computer to be simulated are determined by parameters which are read in as data by the simulator. Once these are set up, further parameters are used to enter a program coded in the machine code of the simulated computer. The program will be executed with monitor printouts of specified registers or portions of memory.
(Received June 1974)

1. Introduction

A convenient method of communicating the basic principles of operation of a particular computer is by specifying its semantics in some suitable notation. Not only does this provide the receiver with the essential information in a conveniently condensed form but it also causes the person providing the information to think carefully and to crystallise his own ideas on the exact manner of operation of the particular machine.

This technique is thus useful when one is studying the basic manner of operation of a computer in courses such as machine architecture, or as a background to the hardware required for certain aspects of operating systems. However, if one is to use such a system for teaching, it should preferably be both simple and powerful. For example, one could express the semantics of a computer in a system such as VDL (Lee, 1972); but while this is powerful, it lacks simplicity and is certainly not the most convenient system for this type of application.

The present paper describes a simulator which employs a 'language' (semantic notation) that is both simple and powerful. The user provides parameters specifying the characteristics of the particular computer he wants to simulate and the program to be run on this computer. The simulator then simulates the effect of running this program on the specified machine, monitoring the contents of certain registers during the run.

It can thus be used to demonstrate hardware features of a wide variety of existing or hypothetical computers, or to allow students to design computers of their own to meet certain specifications, or to implement their own interpretation of existing computers. The language includes a certain amount of duplication in parts (e.g. input/output) to allow it to be used to teach students at different levels.

The data required by the simulator can be divided into two categories, viz.:

(*a*) COMPUTER DEFINITION PARAMETERS—defining the properties of the computer to be simulated, and

(*b*) PROGRAM DESCRIPTION PARAMETERS—defining the machine code program to be run on the simulated computer and providing 'running instructions' to control its execution.

These two types of parameters are described in the following two sections.

2. Computer definition parameters

The first set of parameters, the computer definition parameters, define the structure of the machine in terms of the registers, the machine code instruction set and the instruction set-up/execute cycle. There are also parameters for defining the peripherals and the character set of the simulated computer. The set comprises seven parameter types in all, namely *B*, *D*, *I*, *E*, *O*, *P* and *C* parameters.

2.1. *The BEGIN parameter*

Before a computer definition, the parameter

$$B$$

is used; this causes all pointers to be initialised. It is used since one may want to run several batches of data representing different computers.

2.2. *The definition of registers*

The *D* parameter defines the registers of the computer. Each definition associates a name with a register or set of registers of a particular size as follows:

	$D\ r{:}n$	defines a register r of length n bits,
or	$D\ r{:}n,m$	defines a set r of m registers each of length n bits,
or	$D\ r{:}n,m,p$	defines a set r of p blocks, each containing m registers of length n bits.

The first form is used to define single registers such as accumulators, next-address register, function register, etc. The second form can be used to define main memory, viz. a set of m registers (numbered 0 to $m-1$) of equal length; while the third form is used to define auxiliary memories, such as disc or drum, consisting of p blocks or buckets, each containing m words of length n bits apiece.

More than one definition may be punched on a card, in which case one uses the form

$$D\ d_1; d_2; \ldots; d_k$$

where each d_i is one of $r{:}n$, $r{:}n,m$ or $r{:}n,m,p$. If a definition overflows onto a new card, the symbol '&' in the first column of the new card indicates the continuation. An example is shown in **Fig. 1.**

2.3. *Definition of machine code instructions*

The machine code instruction set (or possibly a subset of the full instruction set) is set up by defining the effect of each instruction on the appropriate registers of the computer. This is done by means of parameters of the form:

$$I\ n{:}s_1; s_2; \ldots; s_i$$

where n is the instruction code in binary (preceded by the symbol '#') or decimal, and $s_1, s_2, \ldots, s_i$ is a sequence of substeps or statements representing the effect of the instruction code n.

Statements may consist of any of the following

(*a*) Arithmetic:	register = expression
(*b*) Data transfer:	MOVE
(*c*) Input:	READ
(*d*) Condition:	IF
(*e*) Unconditional transfer:	GOTO
(*f*) Shift:	SLL, SRL, SLC or SRC
(*g*) Halt:	STOP or ILLEGAL

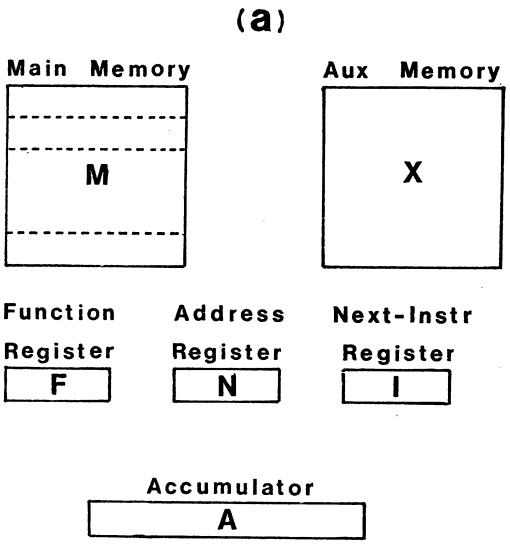


Fig. 1 **An illustration of how the registers of a simple hypothetical computer can be defined using *D*-parameters.**
(*a*) The registers of a simple hypothetical computer: main memory (consisting of 16 8-bit words), auxiliary memory (comprising 16 blocks or buckets, each containing eight 8-bit words), function register, address register and next-instruction register (each of four bits) and an accumulator (16-bits).
(*b*) The corresponding computer definition parameters defining these registers.

(*h*) Execute: EXECUTE
(*i*) Subroutine call: see section 2.5
(*j*) Initiate IO: see section 2.6

Arithmetic

An arithmetic statement is similar to the assignment statement of many high-level languages. The contents of registers (simple or subscripted) and integer constants may be combined by means of +, −, *, / and parentheses in the usual way. An example of an arithmetic statement is

$$A = -(B + C(I))* 2$$

which is interpreted as follows: add the contents of register B and the contents of the register from the set C pointed to by the register I. Multiply the result by two, negate and store in register A.

To access only a portion of a register, the bit positions of the beginning and end of the required field must be specified in square brackets after the register name, e.g.

$$A[1, 5] = E[2, 4] + C(I + 1)[3, 7] .$$

The convention adopted here is that the least significant bit is numbered zero.

If one wishes to propagate the sign bit of the number contained in a register before using it in an expression or storing it in another register, the suffix

@(expression)

Table 1 **Specification of a simple set of machine code instructions for a computer with registers as defined in Fig. 1.**

(*a*) Description of instructions

Machine code function	*Interpretation*
#0000	Load main memory register into acc., propagating sign
#0001	Store least significant 8 bits of acc. into main memory
#0010	Add main memory register into accumulator
#0011	Subtract main memory register from accumulator
#0100	Unconditional jump
#0101	Jump if accumulator negative
#0110	Jump if accumulator is non-zero
#0111	if $N \neq 0$, CALL if $N = 0$, EXIT
#1000	Fetch 8 words from auxiliary into main memory from $M(8)$, then call address 8
#1001	Dump 8 words from main memory to auxiliary memory
#1010	Fetch 8 words from auxiliary and exit to address given in accumulator
#1011	Add into store
#1100	Shift left logical
#1101	Shift right logical
#1110	Read n words into main memory
#1111	Stop

(*b*) The corresponding instruction definition parameters

I	#0000:	$A = M(N)@(16)$
I	#0001:	$M(N) = A[0, 7]$
I	#0010:	$A = A + M(N)@(16)$
I	#0011:	$A = A - M(N)@(16)$
I	#0100:	$I = N$
I	#0101:	IF(A[15, 15].EQ.1), 1; $I = N$
I	#0110:	IF(A.NE.0), 1; $I = N$
I	#0111:	IF(N.EQ.0), 2; $I = A[0, 3]$;
&		GOTO END; $A = I$; $I = N$
I	#1000:	MOVE $X(0, N)$, $M(8)$, 8; $A = I$;
&		$I = 8$
I	#1001:	MOVE $M(8)$, $X(0, N)$, 8
I	#1010:	MOVE $X(0, N)$, $M(8)$, 8; $I = A[0, 3]$
I	#1011:	$M(N) = M(N) + A[0, 7]$
I	#1100:	SLL A, N
I	#1101:	SRL A, N
I	#1110:	READ $M(A)$, N
I	#1111:	STOP

is appended to the register name. The value of the expression specifies the number of bits to be occupied by the result. For example, if one wishes to add the contents of an 8-bit register Z to the contents of the least significant 16 bits of a 24-bit register Y and store the result in a 16-bit register X, propagating the sign bit of Z before addition, one may write:

$$X = Y[0, 15] + Z@(16) .$$

Data transfer

To transfer a number of registers at a time, say from main memory to auxiliary memory or main memory to main memory, the statement

MOVE r_1, r_2, e

is used, where r_1 is a register from a set of registers,
r_2 is a register from the same or a different set of registers,
and e is an expression specifying the number of

registers to be transferred.

For example,

MOVE $C(8)$, $X(0, N)$, 8

transfers the contents of 8 registers starting at $C(8)$ to the 8 registers starting at $X(0, N)$.

Input

The statement

READ r, n

causes n data items to be read from the data cards and stored in registers starting from register r. This is used to obtain the effect of peripheral transfers at a simple level (without using the P parameter).

Condition

To obtain conditional execution of substeps, the statement

IF (condition), n

is used. A condition has the form

expression relation expression

where relation is one of the following: .GT., .GE., .EQ., .LE., .LT. or .NE., and n is a constant. For example,

IF(A[4, 7].NE.B[0, 1]@(4) + C), 3

If the condition is true, then the next n substeps are to be executed; otherwise they are to be skipped.

Unconditional transfer

The statement

GOTO n

where n is a constant, causes control to be transferred to the nth substep of that instruction definition. Similarly

GOTO END

causes all further substeps of the instruction definition to be skipped.

Logical or circular shift

Statements of form

SLL r, e
SRL r, e
SLC r, e
or SRC r, e

are used for left or right logical or circular shifts. In each case r is the register to be shifted and e an expression specifying the number of places it is to be shifted.

Halt

The statement

STOP

halts the simulated computer. The statement

ILLEGAL n

halts the simulated computer and prints a suitable message on the line printer.

Execute

The statement

EXECUTE e

causes e to be evaluated and the result to be interpreted as an instruction code. This instruction code is then executed. Finally control returns to the substep following the EXECUTE statement.

A simple set of machine code instructions is illustrated in **Table 1.**

2.4. *Definition of instruction set-up/execute cycle*

The E parameter is used to define the process involved in fetching an instruction from main memory, resolving the separate fields into separate registers, determining the modified address if indexing or indirect addressing is allowed, updating the next-instruction register and finally executing the instruction. It has the form

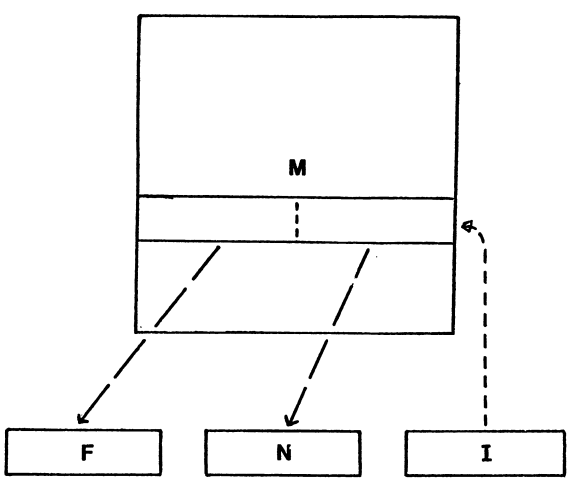
$$E\, s_1; s_2; \ldots; s_j$$

where $s_1, s_2, \ldots s_j$ is a sequence of substeps of the same form as used in instruction definitions. For a simple example, see **Fig. 2.**

2.5. *Definition of subroutines and overflow conditions*

Where sequences of substeps occur repeatedly, subroutines

(a)



(b)

```
E   F=M(I)[4,7]; N=M(I)[0,3]; I=I+1;
&   EXECUTE F
```

Fig. 2 (*a*) **The instruction set-up/execute cycle for the machine of Fig. 1: fetch contents of main memory word pointed to by I and separate function and address components into registers F and N; increment register I and then execute the instruction in F.**
(*b*) **The corresponding instruction set-up/execute parameter for the simulation program.**

```
S   RESET;
S   M(0):#1110100 0; M(1):8; M(3):1; M(4):#11101000; M(5):#10010000; M(6):2
S   M(7):#01000000; I:3
M   1,F,N,I,A
R   5
E   14,#00111111,#00011110,#01011110,#10010000,#01100001,55,8
M   M,1
R   1
E   1,15,#11101000,#10010010,0,#00100011,#00010011,#10000000
M   M,1
R   12
E   6,#00111111,#00010110,#01101101,#11110001,#10001000,#01000111,1
M   R,I
R   78
E   #00010100,15,#00010010,14,#00010011,#10000100,R,0
E   1,#11001111,#01101100,#01001111,0,#00100010,#00010010,#10000101
E   1,#11010001,#01101101,4,#10100010,#00010001,#10000110,0
E   0,#11000001,#00010000,3,#00111111,#00010011,#10000111,1
E   3,#01101100,4,#10100010,#10000100,0,0,0
E   #00101000,2,#00010001,5,#00010000,4,#10100010,0
M   R,F,I,A
```

Fig. 3 Program Description Parameters to set up and execute a bootstrap loader program which loads a binary loader which in turn loads a large program in the hypothetical computer of Figs. 1 and 2 and Table 1.

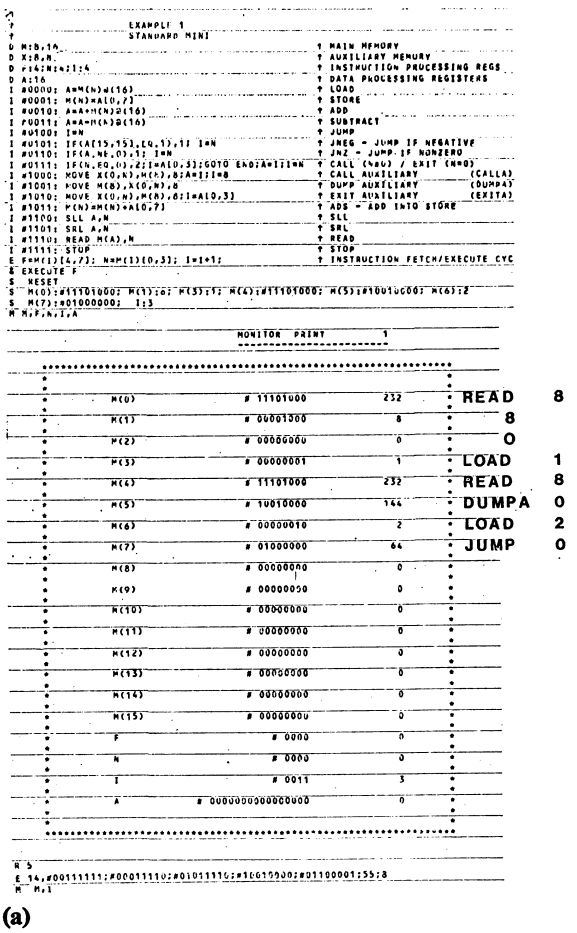


Fig. 4 (a)

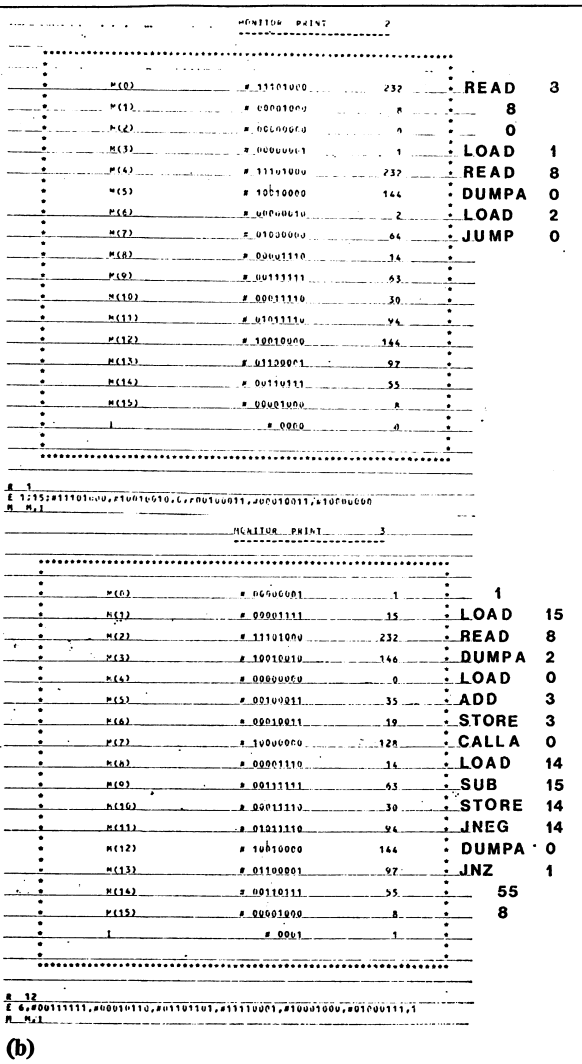


Fig. 4 (b)

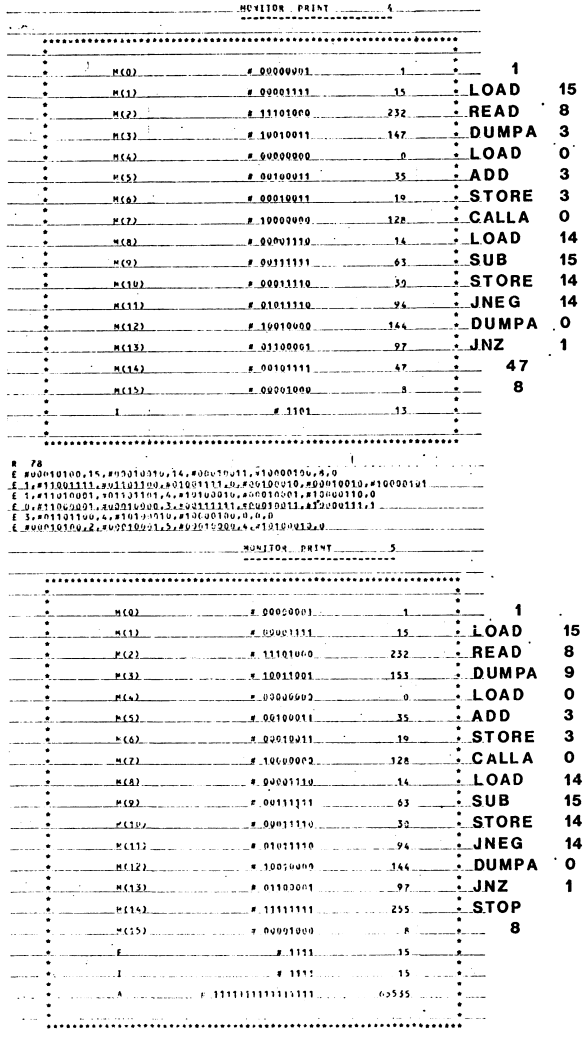


Fig. 4 A simulation run using the parameters shown in Figs. 1 to 3 and Table 1. The contents of registers are printed in binary and decimal form.

may be used to reduce repetition. A subroutine without dummy variables is defined as follows:

$$0\ \%sn : s_1; s_2; \ldots; s_k$$

where *sn* is the subroutine name and $s_1, s_2, \ldots s_k$ is a sequence of substeps which constitutes the subroutine body. A subroutine with dummy variables is defined by a parameter of form

$$0\ \%sn(p_1, p_2, \ldots p_i) : s_1; s_2; \ldots; s_k$$

where $p_1, p_2, \ldots p_i$ is a list of dummy variables, each of form

? letter

Subroutines are called by statements of form

%*sn*	if the subroutine has no dummy variables
or %*sn* (actual parameters)	if the subroutine has dummy variables.

Overflow conditions are treated as a special form of subroutine call. An assignment statement which tests for overflow is written in the form

register ! *sn,r* = expression

where *sn* is the name of the overflow subroutine concerned, and *r* is the register to be set on overflow.

This is interpreted as: evaluate the expression on the right hand side, test for overflow (by calling subroutine *sn*) setting register *r* as required and store the result in the register on the left hand side.

An overflow subroutine is defined with three dummy vari-

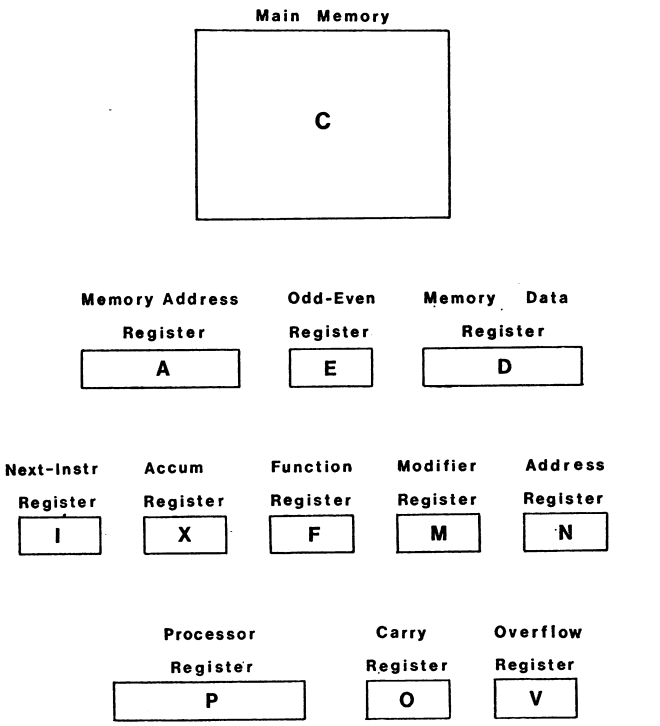


Fig. 5 A simple model of some of the registers of the ICL 1901A.

ables. When it is called, the register on the l.h.s., the overflow register (r) and the value of the expression on the r.h.s. will be substituted as actual parameters in their place.

Examples of subroutines and overflow routines can be seen in Figs. 6 and 8.

2.6. *Definition of peripherals*

To demonstrate independent I/O and processing, details of each peripheral must be specified by means of *P* parameters. These have the form

$$P\ a, c, r, h, n: s_1; s_2; \ldots; s_j$$

where a is the absolute peripheral number (used in the IO statement),

c is the channel number,

r is the name of a single register into which each character being 'read' by the device is stored or from which each character is 'written',

h is a measure of the delay time between requests for hesitations (cycle stealing), which is specified as the number of substeps of the simulated program to be executed between requests,

n is the number of characters in the record to be read or written,

and $s_1, s_2, \ldots s_j$ is a sequence of substeps defining the action to be taken when the peripheral is activated by an

$$\text{IO } a$$

statement in an *I* parameter or a subroutine.

The sequence of substeps $s_1, \ldots s_j$ in a *P* parameter should contain either the statement

INITIATEREAD

or INITIATEWRITE

An INITIATEREAD statement causes control to be returned to the main program sequence, to the substep after the IO statement. After this all substeps executed are counted. When h substeps have been executed, the character read will be stored in the register r and control passed back to the substep after the INITIATEREAD in the peripheral definition. The substeps after the INITIATEREAD could check for parity errors and set the hesitation register. If more characters are to be read, one should branch back to the INITIATEREAD, otherwise exit normally. Similarly for INITIATEWRITE.

An example of this is shown in Fig. 8.

2.7. *Definition of character set*

The parameter *C* is used to define the internal character codes of the computer to be simulated. It has the form

$$C\ \text{“character”}: i$$

where i is the code for the character specified in quotation marks. As with the *D* parameter, any number of these character definitions may be punched on a single card using the form:

$$C\ \text{“}c_1\text{”}: i_1; \text{“}c_2\text{”}: i_2; \ldots; \text{“}c_n\text{”}: i_n\ .$$

3. Program description parameters

This set consists of five parameter types, namely *S*, *R*, *M*, £ and *T*. These control the setup of a program (*S*), the execution (running) of a given number of instructions (*R*), the printing of monitor printouts (*M*), the data required by the program (£) and finally the termination of the simulation (*T*).

3.1. *Setting up a program*

A program is set up by specifying the contents of each register of the simulated computer. The parameter

$$S\ \text{RESET}$$

sets all registers to zero, while the parameter

$$S\ r: n$$

sets up the value n (a decimal or binary number) into the register r. Again, more than one register-value pair may be punched on a card using the form:

$$S\ r_1: n_1; r_2: n_2; \ldots; r_i: n_i\ .$$

3.2. *Executing a program*

To control the simulated execution of the program, one has the parameters

$R\ n$ execute n machine code instructions (or, more strictly, n setup/execute cycles)

and M list print the contents of the registers specified in the list (List items may consist of single register names, names of sets of registers or components of form $r_1 : \text{TO} : r_2$, where r_1 and r_2 are registers of a set of registers),

interspersed with the data parameters which have the form £ $n_1; n_2; \ldots; n_i$ where $n_1, n_2, \ldots n_i$ is a sequence of numbers

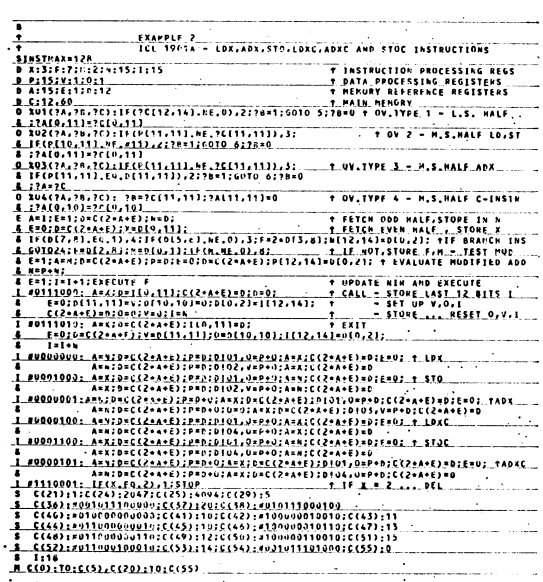
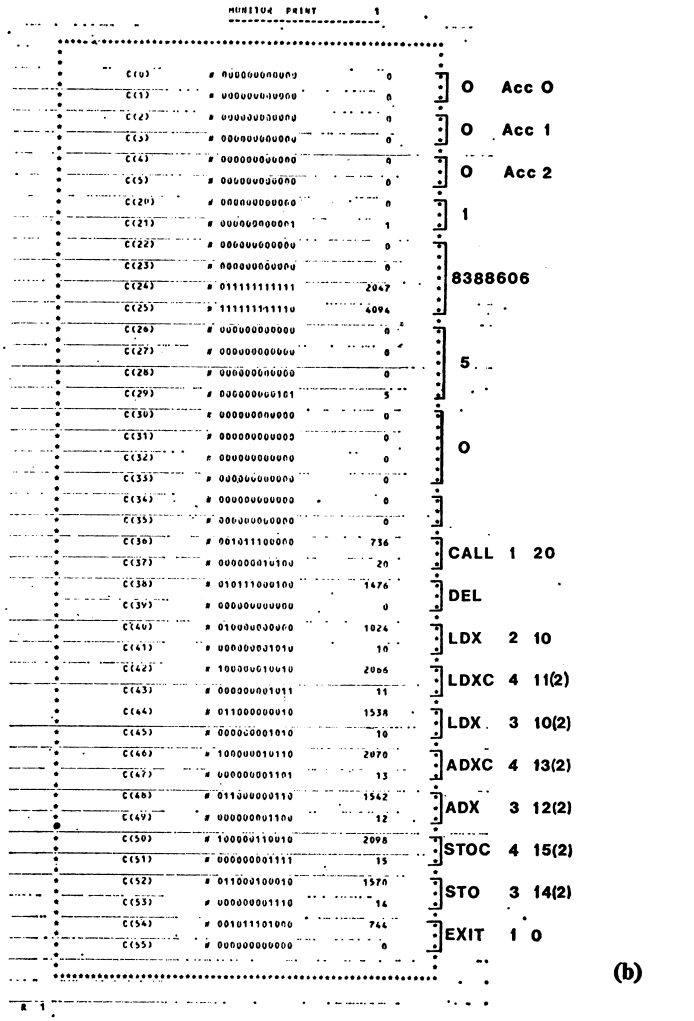
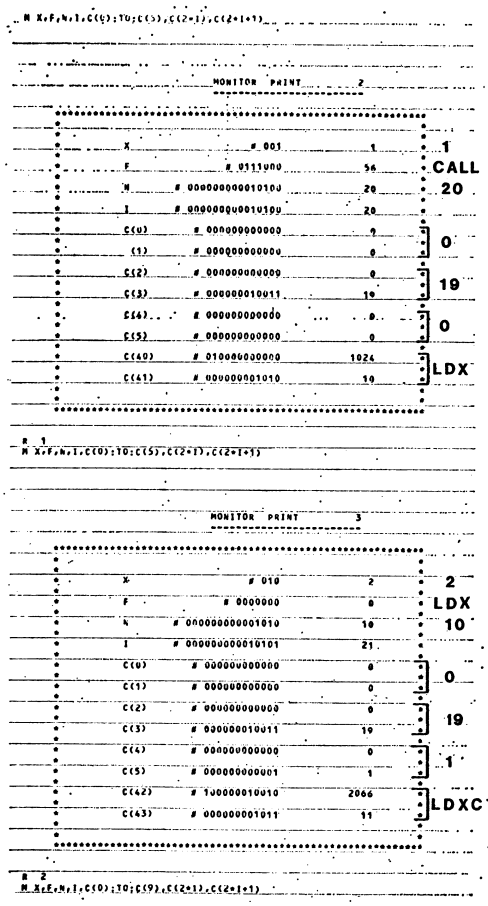


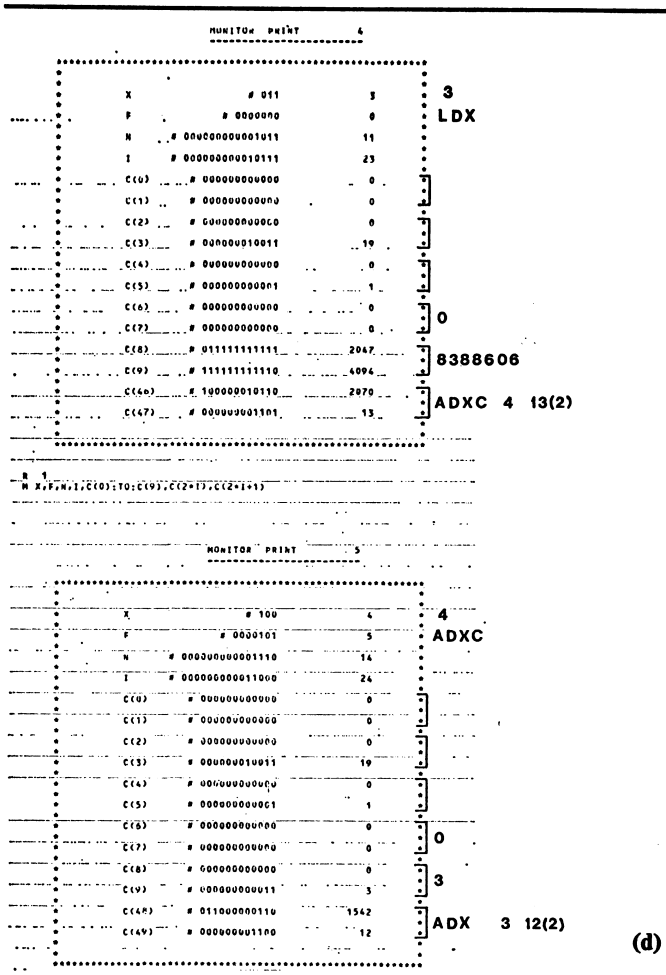
Fig. 6 (a)



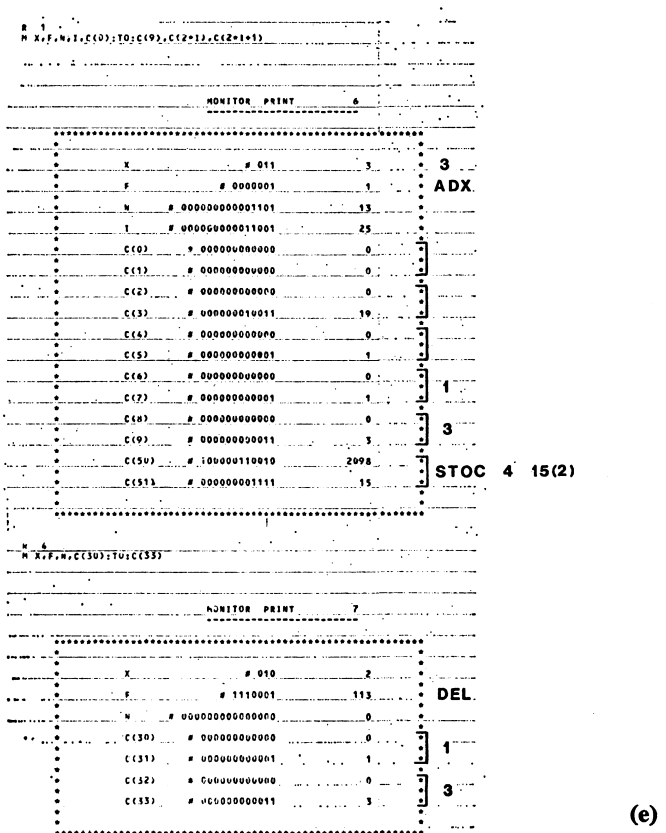
(b)



(c)



(d)



(e)

Fig. 6 A simulation run demonstrating certain aspects of the operation of the ICL 1901A. These include the half-word concept, various types of overflow, different instruction formats for different instructions and relative addressing. The contents of registers are printed in binary and decimal form.

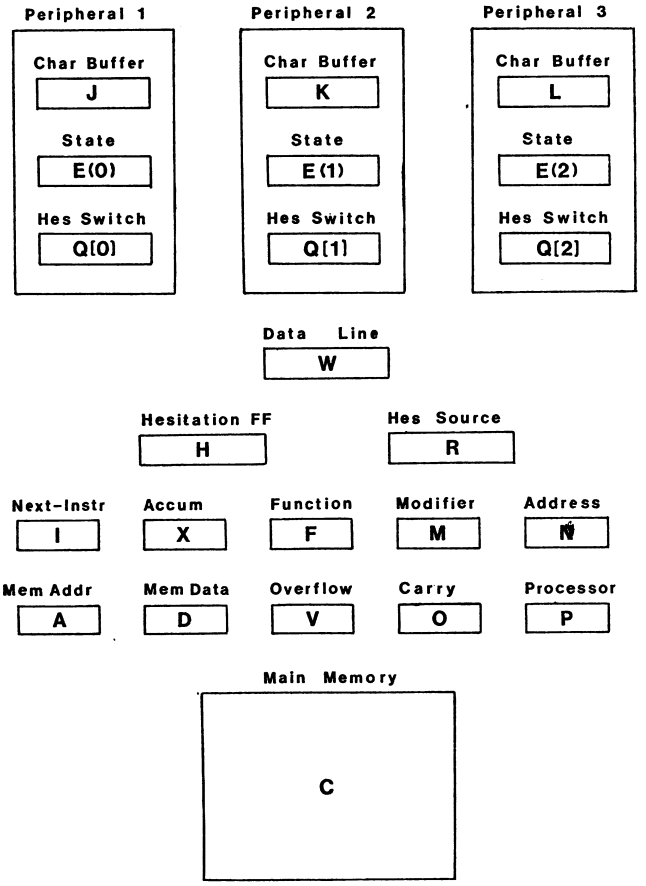


Fig. 7 A simple model of the registers of the ICL 1901A, including three peripherals.

as read from the input device by a READ statement,

and data cards read by an INITIATEREAD statement.

3.3. *Terminating a program*

A program is terminated when a new *B* parameter is read. This announces the start of a new computer definition. Alternatively, the parameter

T

may be used to terminate the simulation run completely.

A typical set of program description parameters is shown in **Fig. 3.**

4. Other facilities available on the simulator

Comments may be included at any point on a card. They are simply preceded by the symbol '↑'. Upon reaching such a symbol, the simulator will ignore the rest of the card.

There are various other facilities which are available on the simulator but which cannot be described here for lack of space. These include tracing facilities controlled by parameters

$TRACE

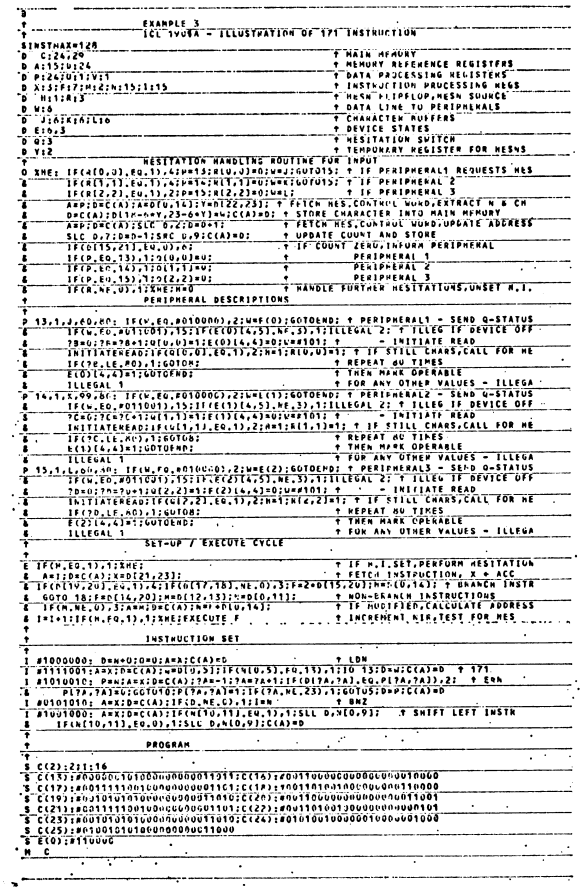
and $NOTRACE

and a set of auxiliary variables which are similar in format to dummy variables.

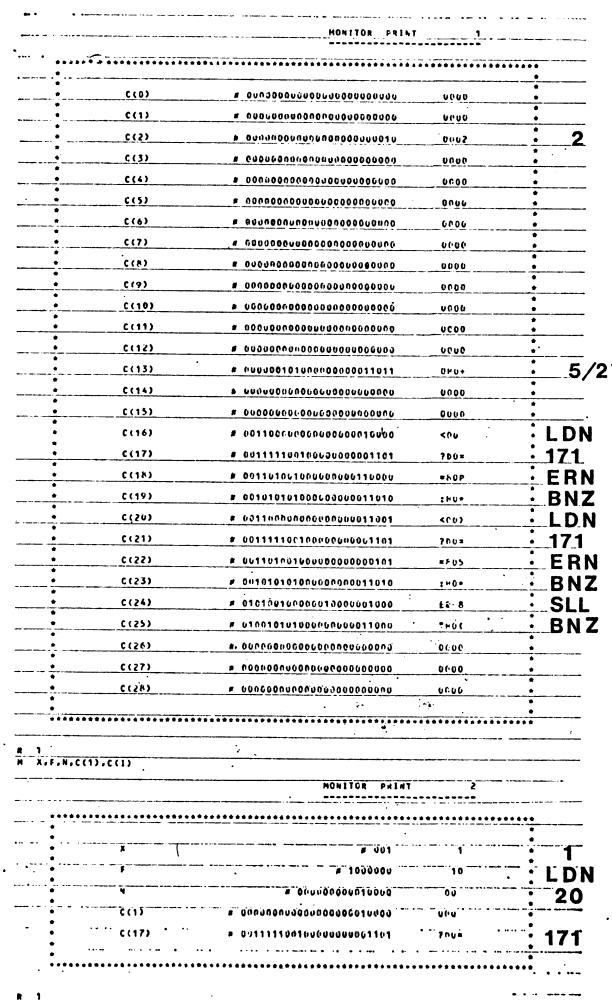
5. Examples

The simulation run for the computer program defined in Figs. 1-3 and Table 1 is shown in **Fig. 4.** This example illustrates the concept of a simple bootstrap loader program loading a more complex loader program which in turn loads the program of interest. This is done within the framework of a very simple machine.

The next example demonstrates some simple aspects of the



(a)



(b)

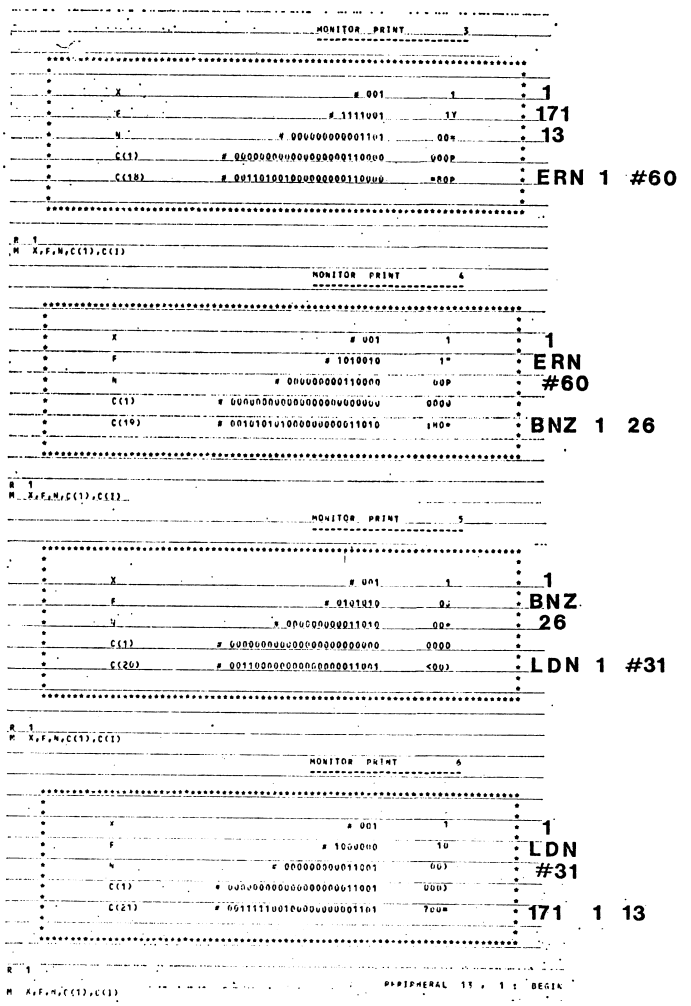


Fig. 8 A simulation run demonstrating how the 171 instruction (an Executive mode instruction) is used first to test the state of a peripheral and then to initiate a peripheral transfer on an ICL 1901A. This is a 24-bit word machine, which actually works on a half-word principle. Furthermore there are two different add instructions for adding from store into an accumulator:

ADXC (used for adding together the least significant halves of two double-length numbers) always leaves the most significant bit zeroised;

ADX (used for adding together the most significant halves of two double-length numbers, or for adding together two single-length numbers) retains the most significant bit, whether 0 or 1.

This results in three different types of overflow condition:

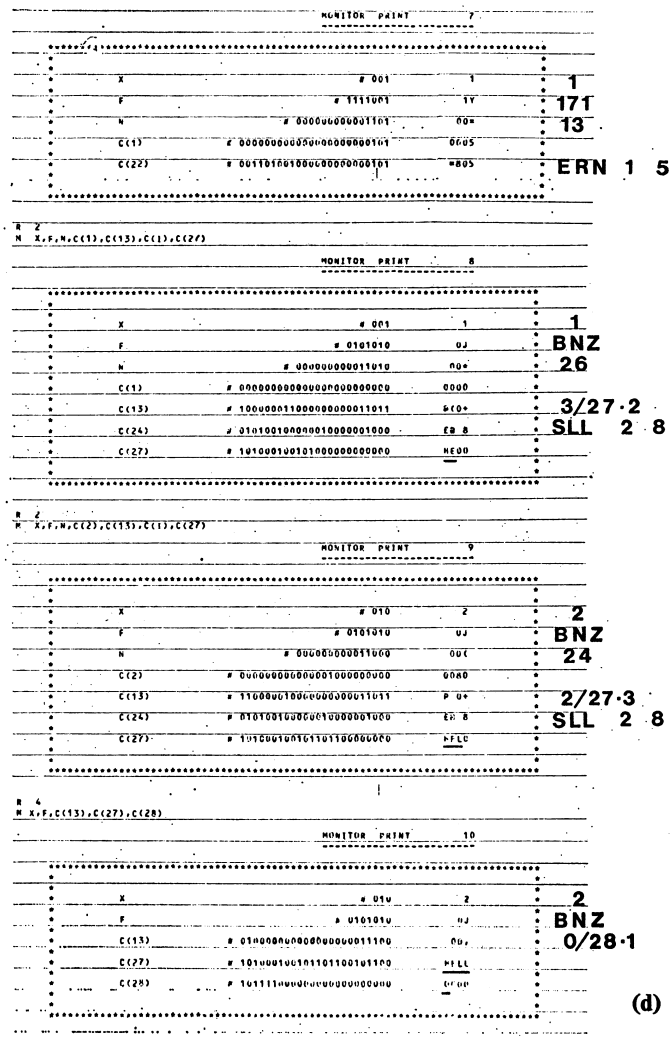
(*a*) on adding the least significant 12 bits of any two 24-bit words (type 1)

(*b*) on adding the most significant 12 bits of the least significant halves of two double-length numbers (type 4), and

(*c*) on adding the most significant 12 bits of the most significant halves of two double-length numbers (type 3).

Besides the two add instructions, the 1901A also has two load instructions (LDX and LDXC), two store instructions (STO and STOC), etc. A simple model of this aspect of the 1901A and a program to illustrate it, are shown in **Figs. 5** and **6.** This model also demonstrates the idea of relative addressing, the implementation of branch type instructions (CALL and EXIT) and of different instruction formats for different types of instruction.

The final example illustrates how the simulator can be used to demonstrate certain hardware features of a machine as might be useful in a course on operating systems. The example



(d)

ICL 1901A. The handling of hesitations is also shown. The contents of registers are printed in binary and character form.

(Figs. 7 and **8)** demonstrates the operation of the 171 instruction within Executive for an ICL 1901A computer. This instruction can be used either to test the status of a peripheral or to initiate a transfer on the peripheral. In this example, the program first calls for the status of the peripheral and checks that it is equal to 60_8, and then initiates a transfer. The reply is checked to see whether the command has been accepted by the peripheral, and the program (Executive) then continues, with hesitations (cycle-stealing) being handled at fixed points in the set-up/execute cycle. The incoming characters are stored in the correct locations and the count is updated on each hesitation (as seen in Monitor Print 8 to 10).

The example can be modified quite simply to illustrate the effect of two or more peripherals operating simultaneously. The streams of characters to and from different peripherals are handled by the simulator while the effects of crisis times can clearly be seen if the hesitation rates (*h*) in the *P* parameters are too small.

Other examples have been run on the simulator illustrating a host of other machine concepts; however, most of these examples are too large to print in a paper such as this.

After submitting this paper our attention was drawn to the similarity of our notation to that of ISP (Bell and Newell, 1971).

6. Conclusion

This simulation program, written in FORTRAN and PLAN, allows one to simulate the functioning of a variety of actual or theoretical computers. It is useful in teaching basic concepts of machine architecture and in illustrating hardware design con-

cepts necessary to understand operating systems—for example, hesitations, interrupts, various addressing techniques (such as relative, indirect and two-component addressing—with page and word registers), storage protection techniques such as base-limit or protect key systems, privileged and unprivileged modes, program status word, interrupt priorities, handling of multiple interrupts, stack machines, etc. In general, we have found it an extremely useful aid to teaching these aspects of computer science.

References

BELL, C. G., and NEWELL, A. (1971). *Computer Structures Readings and Examples*, McGraw-Hill.

LEE, J. A. N. (1972). *Computer Semantics*, New York: Van Nostrand Reinhold Co.

Book reviews

Computer Aided Control System Design, 1973; 244 pages. (IEE Conference Publication No. 96, £8·30)

This publication consists of a set of papers presented at an IEE Conference on Computer Control System Design, 2-4 April 1973, and represents the state of work in this field in the UK at that time. The papers can be split into roughly three sections: (i) identification and modelling, (ii) design, and (iii) simulation, which correspond to the three main activities of the design engineer. The number of papers which fall into each of these categories are, respectively, eight, nineteen and three.

The papers on identification and modelling include descriptions of three comprehensive packages of interactive programs for identification (Clarke, Shellswell and Young, Goodwin *et al*), the latter incorporating an optimal test signal design method. The remaining papers in this area are concerned with various techniques for the reduction of high order system models, a survey of the field being given by Towill. An interesting discussion of some practical modelling problems in relation to steel rolling mills is given by McClure.

Several papers describe comprehensive interactive design program packages and CAD techniques for: (i) linear single input, single output (siso) systems (Allen and Atkinson, Shearer *et al*, Webb, Woodward and Daly), (ii) nonlinear siso systems (Gray and Savvides), (iii) linear multivariable systems (Belletrutti, Fallside *et al*, MacFarlane, Mayne and Chuang, Munro and Ibrahim, Seraji, Young *et al*), and (iv) optimal control (Brown, Burt, Elkin and Daly, Healey and Jones, Mayne, Mobley and Paddison, Weislander). Clearly these areas could be further subdivided to display their specialisations. Two noticeable unifying features are the extensive use of interactive programming techniques, and the importance of graphical presentation of results, features which will undoubtedly form the basis of all control systems CAD programs in the future.

Simulation methods in design are described by Harris and Miles, and Revett. A survey of continuous system simulation languages, with particular emphasis on their industrial usage, is given by Gulland.

As a whole this set of papers describes, or makes reference to, most of the work done in CAD in this field in the 1970-73 period. There is a nice balance between papers which describe successfully implemented packages, and those concerned with future projects. Implicit in their descriptions are the essential features of CAD programs in this area: interactive, command driven, graphical. The only noticeable shortcoming is the absence of any papers which describe real problems which have been approached and solved making use of CAD programs. This can be taken as an indication of the state of acceptance of these design tools by industry, a situation which one would hope will be improved in the future. Another important feature which is absent from many papers is a deep appreciation of the numerical problems implicit in many of the proposed algorithms, especially in relation to implementation on machines with short word length.

M. J. DENHAM (London)

Human Congenital Malformations, the Design of a Computer-aided Study by E. Gal and I. Gal. 1975; 194 pages. (*The Butterworth Group*, £7)

This is an excellent book in almost all respects except its unfortunate title, in which the emphasis is quite misleading. The object is to explain how to design and carry out computer-aided studies and surveys. Since the authors took part in one such study on human congenital malformations, it is used partly as an example to illustrate the points: but other examples are used too.

The central theme is how to make sure that data obtained for future analysis, is useful, relevant and above all completely reliable. Thus the forms or questionnaires on which the information is recorded should be both as acceptable as possible to the specialist in the field and easy for the card-punch operator to use. It is essential to have advice from experts to know what information is useful and relevant. It is equally important to have the questions vetted by laymen and others to make sure that they are easy to understand, unambiguous, and likely to result in truthful answers. Methods of checking the accuracy of the information in various stages (when it is obtained and recorded, when it is punched, and when it is processed) are outlined. The emphasis is on seemingly simple and commonsense precautions which can be easily and disastrously overlooked. For technical details, such as programming or statistical methods, the advice is to consult professional programmers and statisticians.

Anyone but the most seasoned expert considering a survey type of investigation in any field (medical, psychological, social) would be well-advised to read this book first.

A. B. SMITH (London)