EDUCATIONAL MINI-ACCESS ON AN ELLIOTT 803

William Tagg

May, 1970.

ABSTRACT

This research has been centred on providing an on-line system for teaching basic computer concepts to beginners. The approach has been via a hypothetical computer which has been simulated on an Elliott 803 at The Hatfield Polytechnic. Three simultaneous users of the system are connected locally or via the GPO datel 200 service to the computer, the whole system operating in an interpretive mode.

This has not been a piece of pure research. Indeed the Polytechnic has encouraged the project as a pilot scheme for its plans to install computer terminals into a number of schools and colleges as well as within the Polytechnic itself.

The work is believed to be unique in the following respects:

a) It is the first time that an on-line computing system has been provided via software written especially for school use.

b) It is the first time that multi-access has been attempted on an Elliott 803.

c) The hypothetical computer has a number of unique design features which provide interesting programming possibilities and unusual diagnostic capabilities.

d) Compilation of high-level statements into the machine code of the hypothetical computer takes place in such a way that the student is continually made aware of many of the techniques and associated problems.

C O N T E N T

ACKNOWLEDGEMENTS

The implementation of BBC3 at Hatfield has depended for its success on a large number of people. The following in particular should be mentioned:

Mrs. Jennifer Aellen who wrote a large part of the system and on whose debugging techniques the whole project relied

Mr. Allan Croxon who designed, built and maintained the multiplexer

Mr. Gerald Fletcher who took over from Mrs. Aellen and who has written some of the supporting software

Mrs. Sydney Hassall who has used BBC in the classroom from the start and who has provided so much constructive feedback

Professor Bryan Higman who has provided much of the inspiration

Mr. Benedict Nixon who started it all

Finally, I should like to thank the Hatfield Polytechnic for financial support.

BIBLIOGRAPHY

C and G 319 Examination Regulations (available from City and Guilds Institute)

TAM specification (available from S.C.A.P.E.)

The IMDAC programming manual by A.J.T. Colin (University of London Institute of Computer Science)

CES course work (available to those who buy the CES teaching package from ICL)

Working papers on SIMULAC (not published)

A first course on the Schools Computer by D.M. Taub and J.D. Tinsley (published by IBM UK Lab Ltd. Winchester)

Argus 600 User Manual (Ferranti Ltd., Wythenshawe, Manchester)

NEC correspondence course in Logic and the Computer (National Extension College, Cambridge)

BBC Booklet Maths in Action Part 2 (BBC publications)

A Mini-access System for Schools by W.Tagg (BCS Educational Year Book 1969-70)

A conversational Computing System for Educational Use by G.M. Bull (Ph.D Thesis, London University 1968)

Computers in U.S. High Schools by W.Tagg (Computer Education, Volume 2, April 1970)

Computers for all (A suggested syllabus published by the Schools' Committee to the BCS)

The Outer and Inner Syntax of a Programming Language by M.V. Wilkes (Computer Journal Volume II, No.3)

Tel-comp2 User's Manual (Time Sharing Ltd.)

Dartmouth Basic 5th Edition (Published by Dartmouth College U.S.A.)

Basic Machine Programming by J.K. Iliffe (published by MacDonald)

1. INTRODUCTION

BBC3 is a term used to describe an on-line system developed for an Elliott 803 at The Hatfield Polytechnic. Multiaccess use of the computer with three simultaneous users has been established via a multiplexer built by a local electronics firm. A hypothetical computer has been defined and this computer has been simulated on the Elliott. The assembly code of this hypothetical machine is the basic programming language provided, but this is supplemented by certain high level facilities used via a command language within which the assembly code is embedded. This command language, whose syntax and semantics contrast sharply with those of the assembly code, has another main function; it provides a means of real time communication between the user and the computer to enable him to control the system and provide diagnostic help for himself. The way the command language is implemented is not explained to the user; he sees only the results it produces. In fact the whole of the command language, the assembler and certain library routines are written in Elliott machine code. This has been done mainly for efficiency and to enable the best possible use to be made of the 8k core store available, but it would be possible to recode the complete implementation (apart from certain parts of the simulator itself) in the machine code of the hypothetical computer.[1] This would give the system a very large measure of machine independence and would facilitate implementation on other machines.

The system is designed to provide outstations, in local schools and colleges via the GPO datel 200 service, and outstations within the Polytechnic, with a tool for teaching basic computer science. Every attempt has been made to remove from the language (especially the first subset that would be introduced to the student) all

[1] Implementation on the PDP-10 at Hatfield will be largely machine independent.

technical jargon which seems so commonplace in computer circles generally. It has been shown that at its fundamental level, BBC is suitable to demonstrate the basic concepts of a stored program to eleven year old school children. At the same time the computer science undergraduate is able to use the system to investigate the more subtle features of a computer.

Historically, the system originated out of an idea introduced into the television series 'Mathematics in Action: Logic and the Computer' first broadcast during the Spring of 1966. In this series, Benedict Nixon used a simple hypothetical computer and the 'Take and Put' code of this machine formed the basis of an implementation still being used by the National Extension College in a course associated with the TV series. This implementation (BBC 1) was extended to provide conversational access using a single teletype and a simple interface with the computer. This interface allowed serial impulses to and from the teleprinter from and to the least significant bit of the accumulator. Reconstruction of these impulses into characters was by software. The system was used via a land line and on an open shop basis by students of Hatfield School and Mid-Herts College of Further Education. Both these establishments have pioneered 'A' level syllabuses in Computer Science. Hatfield School submitted twelve students for the first time in 1968 for a special 'A' level offered by Oxford Delegacy of Local Examinations and Mid-Herts College has established a course leading to the new AEB Computer Science 'A' level.

The use of BBC 1 in the classroom enabled the definition of BBC3 to be formulated around those ideas which proved most successful in the teaching situation. Conversely, those features of BBC 1 which proved to be less rewarding from a pedagogic standpoint were dropped.

During the 1967-8 academic year, a series of seminars under the general title 'Simsys' was held at the Institute of Computer Science, London University. The purpose of these seminars was to define a hypothetical computer for teaching purposes which would be capable of being revealed to the student as a series of nested Chinese boxes. The student could be led to a more complex structure from simple beginnings. The Simsys seminars have considerably influenced the design of BBC3.

More recently, a working party set up by the United Kingdom Co-ordinating Committee for Examinations in Computer Science to look into the feasibility of recommending common low and high level languages for use in examinations was able to draw on the experience gained from the implementation of BBC3.

It should be emphasised that the decision to include or exclude a particular feature in BBC3 was made on educational grounds only (easily possible in an interpretive environment) with the result that the final order code, although devoid of the exceptions which muddle the beginner, contains a certain redundancy. This redundanoy has been deliberately exploited to provide good diagnostic aid for the user. Diagnostic aid is frequently added to such systems using techniques that are outside the design of the hypothetical machine. Here, this has been avoided at least as far as the student is concerned, not just because of the difficulty associated with explaining this type of 'magic' to the more thoughtful student, but because in the event it proved unnecessary. The design of the machine, the construction of the assembly code and formulation of the command language have evolved as a whole. It has been recognised that certain desirable features could have been introduced as part of the 'hardware' of the computer or via 'software'. In each case, the decision to include or exclude a particular 'hardware' feature has been tested on the

ability of the software to compensate adequately either through the assembly routine or the command language.

No-one would describe machine code (even this machine code) as convenient for anything and school children, who are probably prepared to put up with more than most, need to be shown some of the fascinating features normally associated with high level languages. To overcome this objection, the command language has been extended to compile certain high level statements into the machine code of the hypothetical computer. These compiled instructions (which are added to the end of the student's existing program) are then typed back to the student so that he is able to gain some insight into the intricacies of compilation and has a complete record of his program.

The student is equipped with a set of library routines and a simple filing system[1] which he can use to store several completed or half completed programs. He can suspend a program during its running stage, file it and at a later date retrieve it and allow it to continue from where it left off.

The student who wishes to use the computer as a tool is able to use the filing system to call by name (on a read only basis) the more common programs that one would expect such an installation to provide. Such programs are stored with their own documentation and a general file giving information about the whole of the library is available.

[1] The filing system, as implemented on the 803, relies on the very slow Elliott magnetic film. Concessions to existing hardware have limited the design of the command language in this respect.

Naturally, the system is equipped with a full set of diagnostics and monitoring facilities. Because of this, it has been found that students using the system have required so little help from the staff that the teachers have tended to lose track of their progress. To overcome this difficulty, a statistical summary of the student's progress is output at the end of each session. This summary can only be output by the operator at the centre.

Students learning to program for the first time need to be made aware of different ways in which the content of a store location can be regarded. Too often they gain the impression that part of the store contains 'numbers' and part contains program. The student of BBC3 is forced to consider the various ways in which a location is to be regarded because of the type arithmetic which is associated with the machine code. Each cell carries two extra bits which normally play no direct part in the arithmetic but which, when the cell is called as an operand, are used to modify the function. The four different kinds of word defined by these type bits are

Integer words
Floating point words
Program (or instruction) words and
Strings.

Because of these type bits, it is possible to list a student's program from the machine code version alone, obviating the need to store a separate source image of each program.

Probably the biggest gain that results from the type arithmetic is that it enables the computer to be defined in such a way that fixed and floating point arithmetic can exist side by side. This means that it can be used as 'Imdac' is used on the one hand, and as the City and Guilds Mnemonic code is used on the other.

2. TEACHING BASIC COMPUTER SCIENCE WITH A HYPOTHETICAL COMPUTER

Although the concept of studying a hypothetical computer rather than a real one is popular in many of the Universities and research establishments, the question still has to be answered 'Why study the hypothetical rather than the real?' There are a number of different answers to this question depending on where the questioner stands.

Iliffe answers the question for the computer designer. He suggests that current machine design, programmer practice and selection of programmer personnel and problems for solution is a self perpetuating situation creating only those needs that it is able to satisfy. It is only by making a clean break with tradition that the design requirements of a computer for today and tomorrow can be clearly seen.

Another standpoint is that of the educationalist. Here the requirements are more rigid, for apart from specialist courses, the need is to present to the student the facts as they are rather than (possibly) as they should be. What better, one might ask, than the study of a real machine? It is this question which needs answering in detail, for the work undertaken in this project is only valid if the answers to this question are valid:

a) It is possible that the hypothetical machine simulated on a real machine is more typical of computers in general than the real machine on which it is simulated. Currently, at Hatfield where one has the choice of teaching students about the Elliott computer or the BBC3 machine, many points arise where the imaginary is more typical than the real. A few are listed below.

   i. More typical index modification

   ii. Indirect addressing is possible.

iii. One instruction per word rather than two

iv. Interrupt facilities exist

v. Input/output is autonomous.

b) The design of the hypothetical computer can allow greater freedom to introduce topics into the syllabus in an order dictated by educational requirements rather than by necessity. One can allow the student more freedom to make mistakes, particularly in a multi-access environment where it is possible to hand over the whole machine to a number of different users simultaneously.

It is helpful at this point to introduce an analogy. Suppose that it was economic to provide the learner driver with a hypothetical car. He might have a set of dummy controls which he could use as if they were real and his actions would control a video display of what he might see on the road in front and behind through a 'mirror'. The instructor would be in a position to present traffic situations as he thought appropriate. He might decide that the starting and moving off sequence was difficult and should not be introduced at the beginning of the course, so that the student might start a lesson already in top gear and on a motorway (illegal in practice for learner drivers). From the point of view of getting from one place to another, the hypothetical car is useless and no-one would suggest that it should replace a real car for the student. It might, however, save petrol, dents and scratches.

The hypothetical computer, by comparison, actually produces results. Theoretically results are produced more slowly since its speed is typically slower by a factor of a hundred or more than the machine on which it is simulated, but in practice the total load on the CPU can be less, since the run time of a program

is such a small fraction of the total development time.

c) The computer on which the hypothetical machine is simulated can undertake diagnostic work for the student or teacher. It can provide a helping hand to the student with more freedom than is sometimes possible with either a real machine by itself or the hypothetical machine by itself. For the teacher a statistical summary of student progress can be more comprehensive.

d) The hypothetical computer is more versatile than the real machine. In a teaching environment it is easily possible to afford several hypothetical computers (or variants of the same machine) for different levels of courses. Furthermore it is easy to change the design to meet chaning circumstances.

All that has been said so far assumes that certain standards of honesty are maintained about the implementation of the simulator. Ideally the simulator should be written as a program within the parent machine, to process as data an image of the hypothetical store within the real machine, so as to produce results consistent with the design of the hypothetical computer. The way in which this image is kept within the real machine is only a matter of convenience. It does not matter if the format is different within the hypothetical computer provided each is uniquely deduceable from the other. Programming languages and other associated software should be within the hypothetical machine so that the student can see them as just other programs treating his program as data. In practice, we may allow the implementation to depart from these requirements provided the following safeguards are met. The user must be unaware of the fact that he is programming the hypothetical machine rather than its hardware realisation. (We might feel inclined to relax this requirement as far as speed of

operation is concerned.) This implies that programming languages (the protected sort) can be implemented on the parent machine provided their implementation remains feasible in the hypothetical machine. In practice, this relaxation provides a very large saving in time since it is at the translation stage that most students make heavy demands on the CPU.

It has been suggested that the machine code instructions of the hypothetical computer should be capable of being compiled into those of the real machine, that such a solution would speed up the running of a program compared to the interpretive system implicit in the type of simulation described. It is the author's belief that any such realistic attempt must violate the honesty requirement stated above.

3. FORMAL DESCRIPTION OF THE MACHINE

In this chapter, the definition of the single user version of the BBC machine is expressed in terms of an ALGOL simulation program. It is not expected that this simulation would ever be used as a working model of the computer and in any case such a simulation would only become possible if the machine on which it were to be run could handle sufficiently large variables of type real and integer, and variables of type real to a sufficient degree of accuracy. Only those parts of the BBC machine which merit a detailed study are given and in any case the reader is advised to study the ALGOL in conjunction with the less formal description of the machine given on pages 53 to 60 in the user's manual (appendix 5) and with the notes which follow the ALGOL in this chapter.

AN ALGOL DEFINITION OF THE MACHINE

```
    begin integer CIR,acc operand,store operand,I,F,P,S,
                  zero,acc1,acc2,remainder,SCR,overflow,
                  tolerance,link,eflag;
          Boolean Dmode;   integer array store[0:1023];
          switch function:= nthg,add,subt,mply,dvd,take,
                  mod,clr,and,or,neqv,not,shfr,cycr,oput,
                  iput,put,incr,decr,type,chyp,exec,libr,
                  jlik,jump,jaz,jnz,jlz,jgz,joi,ldn;
A   integer procedure field(cell,top limit,bottom limit);
          integer cell,top limit, bottom limit;
          begin integer i,temp;
                  temp : = 0;
               for i:=bottom limit step 1 until top limit do
                  if cell÷2↑i ≠ 2*(cell÷2↑(i+1)) then
                  temp := temp + 2↑(i-bottom limit);
               field : = temp
          end of field;
B   procedure replace(cell,top limit,bottom limit,new value);
          integer cell,top limit,bottom limit,new value;
               cell:=cell+2↑bottom limit*(field(new value,
                  top limit-bottom limit,0)-
                  field(cell,top limit,bottom limit));
C   Boolean procedure type(cell,letter);
          integer cell,letter;
               type:=field(cell,25,24)=letter;
D   integer procedure sign extend(word); integer word;
               sign extend:=field(word,23,0) +
                              field(word,23,23)*(-2↑24);
```

```
E | real procedure fl pt(word); integer word;
            if type(word,I) then fl pt:=sign extend(word)
      else if type(word,F)∧((field(word,23,23)=1∧field(word,22,22)=
           ∨(field (word,23,23)=0∧field(word,22,22)=1)
                              then fl pt:=sign extend(
              field(word,23,8)*2↑(field(word,7,0)-271)
      else begin eflag:=19; goto error interrupt
           end operand of wrong type;

F | integer procedure real answer(algol ans); value algol ans;
      real algol ans;
           if algol ans < 0.5*2↑(-128)∧algol ans≥ -0.5*2↑(-128)
              then real answer:=0
      else if algol ans ≥ 2↑127∨algol ans < -2↑127
              then begin eflag:=21; goto error interrupt
                   end floating point overflow
      else begin integer exponent;
                              exponent:=128;
                 again:       if algol ans ≥ 1∨algol ans < -1 then
                                 begin algol ans:=0.5*algol ans;
                                       exponent:=exponent + 1;
                                       goto again
                                 end                              else
                              if algol ans < 0.5∨algol ans ≥ -0.5 then
                                 begin algol ans:=2.0*algol ans;
                                       exponent:=exponent-1;
                                       goto again
                                 end;
                 real answer:=2↑24 + 2↑8*entier(algol ans*2↑15+0.5)
                                 + exponent
           end of real answer;

G | zero:=0; acc1:=1; acc2:=2; remainder:=3; SCR:=4;
  | overflow:=5; tolerance:=6; link:=7;

H | I:=0; F:=1; P:=2; S:=3;
```

```
   comment          INSTRUCTION  EXECUTION  CYCLE;
I  repeat:          store[zero] :=0;
J                   CIR := store[field(store[SCR],9,0)];
                    store[SCR] := store[SCR] + 1;
K        if ¬type(CIR,P)  then
                    begin eflag:=17;  goto error interrupt
                    end No instruction in this cell;
L        if  field(CIR,23,22) ≠  0 then goto monitor interrupt;
                    comment this instruction is to be obeyed
                       interpretively by a special monitoring routine;
M   L1:  if  field(CIR,17,17) = 0 then goto direct;
         if  type(store[field(CIR,9,0)],F)   then
                    begin eflag:=18;  goto error interrupt
                    end F-word for store access;
                    replace(CIR,9,0,store[field(CIR,9,0)]);
N  direct:          replace(CIR,9,0,field(CIR,9,0)+store[field(CIR,21,18)]);
P                   acc operand:=if field(CIR,16,16)=1 then store[acc2]
                                                       else store[acc1];
                    store operand := store[field(CIR,9,0)];
                    Dmode :=    field(CIR,15,15)=0;
Q        goto   function[field(CIR,14,10)-1];
R  normal return:
                    store[field(CIR,9,0)] := if Dmode then store operand
                                                      else acc operand;
                       if field(CIR,9,0) = field(CIR,16,16) + 1 ∧
                          field(CIR,14,10)>15 ∧ field(CIR,14,10)< 20
                          then goto repeat;
                    store[if field(CIR,16,16)=1 then acc2 else acc1] :=
                          if Dmode then acc operand else store operand;
         goto  repeat;
   comment end of instruction execution cycle;
```

```
S  nthg: goto normal return;
    add: if type(acc operand,I)∧type(store operand,I)
         then begin acc operand := sign extend(acc operand) +
                                   sign extend(store operand);
                    if acc operand>2↑23-1∨ acc operand<-2↑23 then
                    begin store[overflow]:=field(sign extend(
                                 store[overflow]+1,23,0));
                          if store[overflow]= 1 then
                          begin eflag:=40;  goto error interrupt
                          end Integer overflow
                    end;
                    acc operand :=field(acc operand,23,0);
              end
         else  acc operand := real answer(fl pt(acc operand)+
                                          fl pt(store operand));
         goto normal return;
    subt: comment similar to add;
    mply: if sign extend(store[overflow])<0∧type(acc operand,I)∧
                                          type(store operand,I)
         then begin acc operand := sign extend(acc operand)*
                                   sign extend(store operand);
                    if  acc operand = 2↑46 then
                                  begin eflag:=40; goto error interrupt
                                  end Integer overflow;
                    store[acc2] := field(acc operand,46,23);
                    acc operand := field(acc operand,22,0) +
                                   field(acc operand,46,46)*2↑23
              end of double length multiplication
         else if type(acc operand,I)∧type(store operand,I)
         then begin acc operand := sign extend(acc operand)*
                                   sign extend(store operand);
                    comment similar to integer add from here
              end
         else acc operand:=real answer(fl pt (acc operand)*
                                       fl pt (store operand));
         goto normal return;
```

```
dvd:  comment similar to mply except that division by zero
            is trapped separately;

take: acc operand := store operand;  goto normal return;

neg:  comment similar to subtraction;

mod:  if field(store operand, 23,23) = 1 then goto neg
      else if type(store operand,I)\/type(store operand,F)
           then goto nthg else begin eflag:=19; goto error interrupt
                                end operand of wrong type;

clr:  acc operand := 0;  goto normal return;

and:  begin integer i,temp;
            temp := field(acc operand,25,24)*2↑24;
            for i:=0 step 1 until 23 do
                 if field(store operand,i,i)=1∧field(acc operand,i,i)=1
                 then temp := temp + 2↑i;
            acc operand := temp;  goto normal return
      end of and;

or:   comment similar to and;

neqv: comment similar to and.  Condition is:-
            if field(acc operand,i,i) ≠ field(store operand,i,i)
            then etc;

not:  begin integer i;
             acc operand := field(store operand,25,24)*2↑24;
             for i:=0 step 1 until 23 do
               if field(store operand,i,i)=0 then
                  acc operand := acc operand + 2↑i;
            goto normal return
      end of not;
```

```
shfr: if ¬ type(store operand,I) then begin eflag:=19;
                                        goto error interrupt
                                        end operand of wrong type;
      if store operand ≤ 24 then goto right;
      if sign extend(store operand) ≥-24∧field(store operand,23,23)=1
     then goto left;
      eflag:=23;  goto error interrupt;
      comment operand out of range;
      right:  acc operand:=field(acc operand,23,0)+2↑store operand
                          +field(acc operand,25,24)*2↑24;
              goto normal return;
      left:   acc operand:=field(acc operand*2↑(-sign extend(
                          store operand)),23,0)+field(acc operand,
                          25,24)*2↑24;
              goto normal return;

cycr: if  type(store operand,I ) then begin eflag:=19;
                                        goto error interrupt
                                        end operand of wrong type;
      begin integer temp;
            temp:=if field(store operand,23,23)=0 then store operand
                  else 24+sign extend(store operand);
            if temp>24∨temp<0 then begin eflag:=23;
                                   goto error interrupt
                                   end operand out of range
            else acc operand := field(acc operand,23,temp)
                               +field(acc operand,temp-1,0)
                               *2↑(24-temp) +field(acc operand,
                               25,24)*2↑24
      end;
            goto normal return;

oput: if type(store operand,P)∨ type(store operand,F)
      then begin eflag:=19;  goto error interrupt
           end operand of wrong type;
      string out(field(store operand,5,0));
              goto normal return;
```

```
iput:  store operand:= string in +3*2↑24;
           goto normal return;

 put:  store operand:= acc operand;
           goto normal return;

incr: comment add 1 to store operand.Otherwise as add;

decr: acc operand:= store operand; comment subtr. 1 from
           store operand.Otherwise as subt.;

type: acc operand:= field(store operand,25,24);
          goto normal return;

chyp: store operand:=field(store operand,23,0)+field(acc operand,
                    1,0)*2↑24;
          goto normal return;

exec: if ¬type(store operand,P) then begin eflag:=19;
                                     goto error interrupt
                                     end operand of wrong type
      else if field(store operand,14,10)=22 then
                                     begin eflag:=3
                                     goto error interrupt
                                     end exec(exec) is illegal
           else begin CIR:= store operand;
                goto L1
                end;

libr: begin switch libr number:= sqrt,ln,exp,read,print,sin,cos
                  tan,arctan,stop,line,int,frac,float,captn;
           if field(CIR,9,0) > 15 then begin eflag:=24;
                                      goto error interrupt
                                      end libr addr part too big;
           goto libr number [field(CIR,9,0)]
      end jump to appropriate subroutine outside the user's store;

jlik: if Dmode then begin store[link]:= store[SCR]; goto jump
                    end
      else begin store field(CIR,9,0) := store[SCR];
           store[SCR] := store[SCR]+1;
           goto repeat
           end;
```

```
jump: if Dmode then store[SCR]:= field(CIR,9,0)
                else if field(acc operand,25,24)=field(
                        store operand,25,24)
                     then store[SCR]:= store[SCR]+1;
           goto repeat;

jez:  if Dmode then
      begin if acc operand=0 then goto jump
       else if type(acc operand,P)∨type(acc operand,S) then
            begin eflag:=19;  goto error interrupt
            end operand of wrong type
       else if field(acc operand,7,0)≤ field(store[tolerance],7,0)
            then goto jump;
            goto repeat
       end       else
       begin Boolean skip;
             if type(acc operand,I)∧type(store operand,I) then
                  skip:=acc operand =store operand
       else if type(acc operand,P)∨type(acc operand,S)∨
               type(store operand,P)∨type(store operand,S) then
            begin eflag:=19;  goto error interrupt
            end operand of wrong type
       else skip:= abs(fl pt(acc operand)- fl pt(store operand))≤
              2↑(field(store[tolerance],7,0)-128);
            if skip then store[SCR]:= store[SCR]+1;
            goto repeat
       end;

jnz:  comment similar to jez;

jlz;  comment similar to jez but no tolerance is used;

jgz:  comment similar to jlz;

joi:  if Dmode then
      begin if store[overflow]≠0 then begin store[overflow]:=0;
                                           goto jump
                                      end
      end      else
      if field(store operand,23,23)=1 then store[SCR]:=store[SCR]+1;
            goto return;
```

```
ldn:  CIR:=store field(store[SCR-1],9,0) ;
      if field(CIR,17,17) = 1 then replace(CIR,9,0,
                                    store[field(CIR,9,0)]);
      if Dmode then  store[field(CIR,9,0)]:=store[field(CIR,21,18)]
               else store[field(CIR,21,18)]:=store[field(CIR,9,0)] ;
      goto return
end;
```

A: field(cell,top limit,bottom limit)

This procedure accesses part of a cell content. The result is the positive integer described by the binary pattern of that part of C(cell)[1] which starts at the bit position 'top limit' and finishes at the bit position 'bottom limit'.[2]

B: replace(cell,top limit,bottom limit,new value)

This procedure replaces the group of bits described by field(cell,top limit,bottom limit) by a new group of bits described by 'new value'. If the binary representation of 'new value' is too large for this field, it is masked down to size before it is inserted.

C: type(cell,letter)

'letter' is either I,F,P or S (standing for 0,1,2 or 3). 'type' takes the value true if this letter correctly describes the type bits of the C(cell).

D: sign extend (word)

This procedure sign extends the 24 bits of the BBC word (ie, excluding the type bits) to the full length of the ALGOL machine.

E: fl pt(word)

If 'word' is a BBC I-word, the procedure simply converts to the floating point form of the ALGOL machine. If 'word' is an F-word, the change is from the floating point format of the BBC machine to that of the ALGOL machine.

F: real answer (algol answer)

This converts the ALGOL floating point format to its BBC equivalent.

[1] Throughout this thesis and the users' manual C( ) is used to denote 'content of'. Other people have used merely a pair of brackets and although this is convenient, students at school who may only recently have been introduced to traditional functional notation are thought to find the full notation more meaningful.

[2] The bits within a cell are always numbered from right (least significant end) to left (most significant end) from 0 to 25. In this way for integer representation 2^N is represented by a one in the Nth bit position.

G: Special purpose registers are referred to by name rather than by their numerical addresses. This is done merely for readability.

H; Types are also referred to by letter for the same reason.

I: The zero address always appears to be the source of zero.

J: The current instruction register is loaded with a copy of the instruction pointed to by the sequence control register.

K: Only P-words can be obeyed as instructions.

L: If monitor bits are present, the instruction must be obeyed interpretively by the monitor so that additional output can take place.

M: If bit 17 is zero addressing is direct, otherwise (provided an F-word is not accessed) the address field of CIR is replaced by the 10 least significant bits of the cell content pointed to by the old address field.

N: Index modification takes place. field (CIR,21,18) is the address of an index register whose content must be used to modify the address portion of CIR.

P: The current accumulator operand is selected and the X-mode bit is tested.

Q: The function field is used to select the appropriate function.

R: Apart from the jump and skip functions which do not affect operands, the return to the main execution cycle is made here so that the new operands can be loaded back into the user store. Note that the store operand is loaded first, this becomes important when the accumulator's address also appears in the address field. The later functions, where the store operand is changed require special treatment in this case.

S: The list of the BBC functions given here should be studied in conjunction with the order code table on page 56 of the users' manual.

4. DESIGN OF THE COMPUTER FROM AN EDUCATIONAL STANDPOINT

The description of the computer given here assumes its hardware realisation. From the point of view of the student user it is probably best to allow this assumption to continue. He needs to feel secure in the more deterministic atmosphere of a hardware rather than software design. Undoubtedly, its hardware realisation is technically possible, but it is more realistic to suppose that a large part of the design complexities are in the hands of extracodes. The boundary between software and hardware continues to get more blurred and there are those who suggest that the student need never be aware of the distinction, that he should accept the complete system as the entity with which he must deal. Beginners, however, often fail to appreciate that their own program input is merely data to another program already in the store. Only by identifying at least the assembler (or compiler) as software can this important concept be established in the mind of the student.

At some stage in his education the student must be made explicitly aware of the fact that he is timesharing with other users. On the current implementation he very quickly becomes implicitly aware of this, particularly when he is using a local terminal alongside other users. In this situation he tends to lean over his neighbour's shoulder to ascertain why his own response has not been immediate. Initially, however, the average student has enough to worry about without considering the multi-access capabilities of the machine, and it is the single user version of the computer which has been defined here. This is a subset of the informal description of the multi-access version currently given to more senior students of the 803 implementation. Plans for its implementation on the Polytechnic's new PDP-10 allow students to formally study the machine at either level.

Clearly the definition of the machine is independent of any programming languages which happen to have been written or 'supplied by the manufacturers'. References in this chapter to concepts which rightly belong to the definition of the assembler should be regarded by the reader as reinforcing the material in Chapter 5.

The machine is a binary computer and its store consists of 1024 cells each of 26 bits plus an undefined store area not available to the user. These cells are given addresses 0 to 1023 and apart from those with addresses 0 to 15 all have identical properties.

Bits 25 and 24 play a special role and define for the computer the way in which the remaining 24 bits are to be manipulated at run time by the arithmetic unit. From the user's point of view these two bits (called type bits) define the four distinct categories of storage given below:

Type bits	formal name	description
00	I-words	integer representation
01	F-words	floating point representation
10	P-words	stored computer instructions
11	S-words	character strings

It is, of course, up to the user to decide how well these descriptions match his own requirements in a particular situation. A user who wishes to store and manipulate addresses or packed information other than character strings may decline to regard the type bits as significant and this is his prerogative, but it has been found that most applications handle information which roughly corresponds to one of these four categories.

The decision to choose a 24 bit cell length (apart from the type bits) represents a compromise between a number of conflicting requirements. It was felt that word length should be long enough to avoid double length

arithmetic most of the time while still giving reasonable accuracy, but not so long that the student gets lost in a sea of bits. It should possess the ability to break down into its component parts with a well-ordered mathematical tidiness. Some power of two would seem to satisfy most of these requirements, but sixteen bit words are restrictively short and would certainly have meant storing floating point mantissae and exponents separately, while thirty-two are on the long side. Eventually it was decided that twenty-four would be an acceptable compromise (twenty-four also has the advantage of accommodating four six-bit characters).

Type arithmetic had been forced into the design of BBC 1 but had remained hidden from the student. The intention was that it could protect the novice from his own foolishness without him being aware of it. Although this technique proved exceptionally satisfactory to start with, the day came when the student wished to 'beat' the system; he wanted to do those things that he had previously been prevented from doing for some very good purpose. Gradually it became evident that deception had to be very skilful if it was to succeed.

Consideration was given to the concept of an address as a special type possibly related to some base address stored elsewhere. In this way, indirect addressing to any depth could have been established:

ADD N

would have meant add C(N) unless C(N) happened to be an A-word in which case a second store access would have taken place so that the meaning became add C(C(N)). If C(C(N)) still remained an A-word the process of indirect access would have been repeated. This idea was eventually dropped for two reasons. Firstly, it is very easy to get into a tight access loop eg: ADD 100 when C(100) equals the A-word 99 and C(99) equals the A-word 100. Secondly, when access is not direct, it is usual to perform

arithmetic on the address(es) prior to access and this becomes impossible without special arrangements.

The fixed point fraction as a separate type was not included, partly because of its arithmetic similarity to the integer and partly because floating point arithmetic hardware has removed the need for its representation for most purposes. Double length integers were also excluded for although their usage is common today, there is nothing fundamentally different between double and single length working. (It should be noted, however, that a double length product to single length integer multiplication and a double length dividend in division are fundamentally important and these are included.)

A single address machine was chosen mainly for its simplicity. Although at first sight a three address machine sounds attractive, since it easily accommodates basic arithmetic statements of the form

result := operand operator operand

such a machine has the following drawbacks:

a) there is a superfluous address when the function is monadic,
b) since most arithmetic involves manipulation of the same operands in quick succession, it is uneconomic to have to name them all explicitly at each step,
c) it does not conform with current practice,
d) array work becomes very complicated.

A single address machine, on the other hand, is free from these objections but suffers from the restrictions associated with transfer control functions. Near sequence control (relative, conditional jumps using the index field to specify a short backward or forward hop), was a tempting solution for it provides a powerful programming tool. It was abandoned for the following reasons:

a) it leads to errors when patching is done,
b) its non-standard use of the index field is not consistent with a machine suitable for school use.

The more traditional skip instructions that have been implemented provide a useful but less exciting compromise to this problem.

Sixteen functions were considered unduly restrictive, 64 functions place too large a burden on the user's memory and are difficult to describe with pronounceable, jargon-free, short mnemonics. The 32 functions provided are supplemented in two ways; firstly by the modification brought about by operands of different types and secondly by the provision of what has been termed X-mode in place of the usual D-mode. Under X-mode the C(A) and the C(store) are interchanged after the function has been obeyed. The effect of X-mode on a transfer control function is to turn a JUMP instruction into a SKIP instruction. The number of variations available could, therefore, be as great as 1024 but a large number of these are designated as errors. These errors cause the current program to be interrupted and control passes to the error routine (outside the user's slice of store) leaving the user's SCR unaltered, so that he is told exactly at what point the mistake occurred. For those users who wish to go further, two functions are supplied for manipulating type bits.

An attempt has been made to keep the code free from the sort of exceptions which usually bedevil the instruction set of a real machine (often for very good hardware reasons or for the convenience of the specialist). Thus SHFR and CYCR use, for the sake of consistency, C(address) as operand and not the address part itself as is normal. It has been found necessary, however, to introduce two functions which use the rest of the bits in the instruction exceptionally. The LIBR function uses the

address bits to specify the required library function directly (all library functions only use C(A) as operand) and one function, the last one listed in the order code summary, uses the index bits to specify a second operand instead of a modifier.

While it is probably necessary for the student to distinquish between integer and floating point arithmetic and their properties, he should be protected from the tedium of mixed arithmetic. This is after all the philosophy of high level languages such as ALGOL. In BBC3 mixed arithmetic is catered for by the provision of the type bits, so that the result of an arithmetic operation is stored in floating point form unless both operands are integers. The READ routine stores constants and data as I-words or F-words. I-word storage takes place unless a decimal point or subscript 10 or both are present.

Jump and skip instructions, which test equality or inequality, are treated specially when their operands are F-words. Equality is taken to mean within a certain tolerance and C(6) enables the user to specify the value of this tolerance. If the difference between two numbers stored as F-words is so small that its exponent part is less than C(6), the two numbers are regarded as 'equal' in this context. C(6) has a rather arbitrary initial setting of 16 which seems to produce reasonable results most of the time. When the greater and less than JUMP or SKIP instructions are obeyed, the tolerance setting is not used and the instructions mean what they say.

The three special bits in the instruction word designate

a) one of two accumulators
b) X-mode or D-mode (already described)
c) direct or indirect addressing.

Although a system of indirect addressing is largely redundant in a machine so generously provided with index

registers, it has proved a more successful way of introducing array work than the more traditional method of modifcation of an address by the content of an index register. Fundamentally, the two methods are not dissimilar as indirect addressing can be regarded as a special case of index modification where i) every cell is an index register and ii) the base address is always zero. Viewed in this light, it can be seen that the indirect address is a more basic and simpler concept to introduce initially. For the most experienced user the use of indirect addressing and index address modification within the same instruction has interesting possibilities.

In BBC3 a single store access is made, followed by index modification, followed by a second store access, so that C(C(N) + C(R)) is used as operand and not C(C(N + C(R))). The absence of an index register implies modification by the register with zero address, but C(0) = 0 always so that no modification at all takes place.

The ability to change type bits makes arithmetic manipulation of the instruction word a possible but cumbersome procedure. There are some teachers of basic computer science who feel that the correct way to introduce array work is by arithmetic modification. The following sequence of instructions illustrates the technique:

```
100 CLR2 0          address arbitrary
101 TAKE PLANT      C(PLANT)=instr. to be modified
102 CHYP2 1         C(A1) changed to an I-word
103 ADD COUNT       add modifier
104 TAKE2+2
105 CHYP2 1         C(A1) changed back to a P-word
106 PUT 107
107
```

where C(PLANT) is an instruction to be modified by C(COUNT) and planted in location 107 to be obeyed.

The instruction TYPE, CHYP and SKET are adequate enough to allow the exploitation of the type arithmetic, yet sufficiently advanced in concept to discourage their indiscriminate use by the novice. Users have found some unexpected applications for this unusual feature, especially where the information being moved about in the computer does not rightly belong to any one type. By associating different categories of data with different types, it is possible to branch a program by using SKET. On the debit side, some students are inclined to disregard the type of their operand and the error message OPERAND OF WRONG TYPE is one which is often seen. The use of the logical functions in this context is a little confusing, the following table illustrates the difficulty (entries shown in octal the initial letter specifying type).

store operand	function	accumulator operand	accumulator result
(S77007700)	NEQV	(P70077007)	(P07070707)
(I70007600)	AND	(F70034572)	(F70004400)
(P70347077)	OR	(I47370234)	(I77377277)
(P76007010)	NOT	arbitrary	(P01770767)

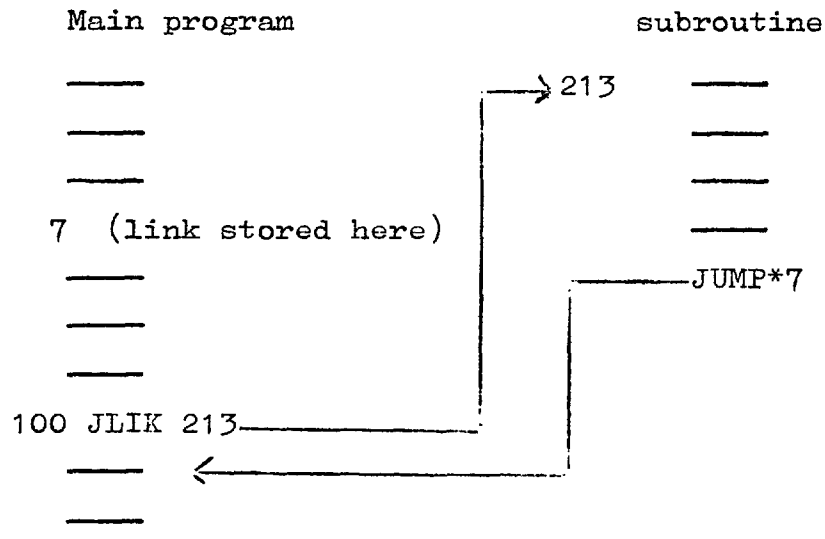
The type bits of the accumulator are not changed by NEQV, AND or OR, but with NOT they may well be.

Link storage is another problem where the final solution has been the best compromise available. BBC 1 used the functions SUBR (store the link and jump) and EXIT (retrieve the link and jump indirectly). The link addresses were in a stack (outside the user's store) so that subroutines could be nested and could call themselves

if necessary. This approach (also used in Elliott Autocode) has a number of disadvantages; the user is compelled to exit from a subroutine via an EXIT instruction otherwise later EXIT instructions pick up the wrong link. This restricts the programmer and if he fails to keep to the rule the errors that result are hard to diagnose. Failing to exit through an EXIT instruction is an error which is difficult to explain to the student who is not brought face to face with link storage. It is also useful to be able to access the link address for for picking up parameters or to dump the link address after an abortive run. Against these disadvantages must be weighed the advantage of simplicity. The creation of subroutines is fundamental and the topic can be introduced early in the course when automatic link storage is possible. Several alternatives exist:

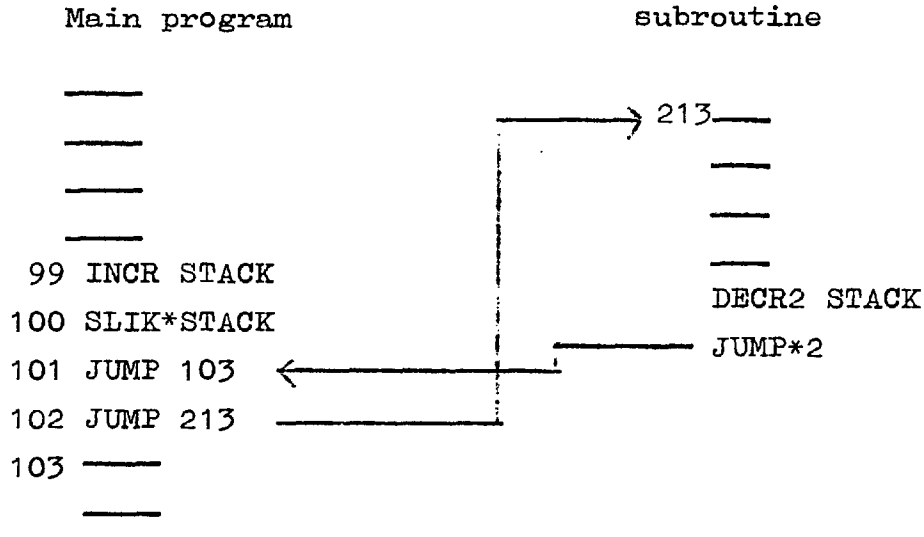
The link address can be planted in the cell prior to the start of the subroutine. This is a common technique and allows subroutines to be nested but does not allow subroutines to call themselves (not a grave disadvantage since links can always be rescued and in any case only advanced students will wish to do this). More serious is the danger that room will not be left for the link so that at run time some instructions could be overwritten.

City and Guilds code provides the user with a function which always plants the link in a fixed location and a student must rescue it even if he wishes to nest subroutines. It is expected that students of BBC will be introduced to subroutine entry and exit by this method. Cell 7 has this special property.



(Note that the SCR is incremented after the next instuction has been accessed but before it has been obeyed.)

SLIK (the equivalent of JLIK in X-mode) plants the link in the specified address and then skips.[1] This provides everything for the more sophisticated user:



[1] A neater subroutine entry and exit is possible if the instruction merely stores the link so that no skip takes place. This change is to be made in the new implementation on the PDP-10.

I-words Designating an I-word by zero in the type bits provides a clear method of giving the student a 'clean sheet' after the 'LOGIN' sequence. The twenty-four bits of an I-word represent integers in the range -2^{23} to $2^{23}-1$ using two's complement arithmetic for negative numbers. The decision to use two's complement rather than sign and modulus arithmetic was made for three reasons:

a) It is much more common in real machines.

b) Although sign and modulus arithmetic is a simpler and more easily understood concept, many students find the study of two's complement arithmetic fascinating and satisfying.

c) The difficulty associated with two's complement arithmetic's least elegant feature, of not being able to change the sign for the smallest possible negative number capable of being represented, is not likely to be met early in the course. The corresponding difficulty of having two representations of zero in sign and modulus arithmetic has to be faced early on.

F-words Type bits 0 1 were chosan so that tests for arithmetic operands could be made by looking for zero at bit 25. The 24 information bits split cleanly into 16 for the mantissa (m) and eight for the exponant (e). The mantissa is stored at the more significant end and uses two's complement arithmetic so that $\frac{1}{2} \leq m < 1$ or $-1 \leq m < -\frac{1}{2}$ and the sign bit is that used for integer representation. The exponent field stores 128 + e in accordance with current practice. Zero is always stored as an I-word.

In a computer with a variable word length it is possible to match the length of the mantissa with that of integer representation. This is convenient when changes from one representation to the other take place.

Consideration was given to the idea of using two words to store an F-word but this would have necessitated separate real and integer declarations. Another possibility would have been to use some kind of floating point representation that had zero exponent for integers, but such a system would have been unrealistic.

S-words Character representation in the machine has nothing to do with any external form of coding. Furthermore, since the system is intended for use on-line, the user need never be aware of the intermediate (eg paper tape) representation of characters.

In choosing a satisfactory character code the following points have been borne in mind:

a) Digits need to be in an ordered sequence so that routines for denary to binary conversion do not require a look-up table.

b) Digits should not be represented by their integer value because i) it is inconvenient to represent the digit zero by the value zero and ii) students find it confusingly easy; they believe that there is no difference between the concepts of character representation and integer representation.

c) Alphabetic characters need to be in an ordered sequence so that problems can be set on alphabetic ordering.

d) Where a whole number of characters fit into the word (a tidy feature anyway) it is helpful for alphabetic ordering if the sign bit is left zero when letters are packed to the most significant end of the word.

e) There must be a combination of bits representing 'no character' and it is necessary for its value to be less than the values of the alphabetic characters.

f) Non-printing characters (with the exception of spaces and new lines) should not be representable [1] and the representation of other characters should be seen as a one-to-one correspondence between their hard copy realisation and their binary representation within the machine.

g) A new line should be one character, not two.

h) Two cases of letters are extravagant and add nothing pedagogically.

i) If a backspace is allowed (this is probably not desirable) it should only be a device for turning one character into another so that '<' followed by 'back space' followed by 'underline' would be regarded by the computer as a representation of the single character '≤' .

j) If is desirable to have a 'character' which is not stored but which causes the last character to be erased. This presents buffer problems:
   i) are two consecutive erase characters possible?
   ii) is it possible to erase a new line? Such as erase character will need a representation for output.

k) There is no need for redundancy.

Index registers and special cells Any of the first 16 cells may be used as an index register. Cell 0, used in this way causes modification by C(0), that is not modification at all since C(0) = 0. Some of the other index registers have special properties. The student will see these as special hardware or software properties and although it is recognised that the distinction between software and hardware is becoming increasingly blurred, some attempt has been made to divide the special purpose registers into the two classes.

[1] Possibly there should be a loophole here. There seems to be a growing need to output a group of ASCII characters for more complex processes than printing messages.

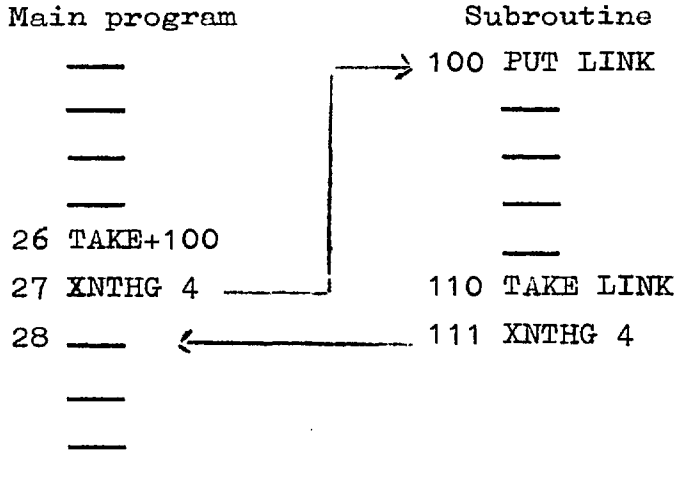
It is hard to see what some of these special properties have to do with index modification and it was tempting to reserve space elsewhere in the user's store. It was tempting also to reserve space for all special cells at the top end of the store out of harm's way, but the difficulty of addressing index registers with large addresses using a four bit field finally caused this idea to be abandoned.

The user is now protected from using the first 16 cells for normal programming by safeguards written into the assembler and the index registers have remained the only cells with special properties, so the unified approach is maintained as far as possible.

In a traditional computer, it is unusual for some of the special purpose cells to be addressable at all, but in BBC3 everything that it is reasonable for the student to have access to is addressable. In particular, both accumulators, the SCR and the overflow indicator can all enter into normal programming procedures. There are a number of other advantages that are consequences of this decision:

a) The transference of information from one accumulator to the other requires no special function.

b) Transfer control functions can be introduced by performing arithmetic on the SCR. The following sequence of instructions illustrates the inherent simplicity in link storage and subroutine entry and exit:



c) Filing a program automatically stores the contents of those special purpose cells so that the mapping to and from film is simple and complete.

Integer Multiplication and Division. For normal work single length multiplication and division take place. It has proved useful to have the remainder available after integer division, for problems on £-s-d and for the binary to denary conversion implicit in the integer print routine. Students have also found it convenient when solving problems on factorisation, prime numbers and modulus arithmetic.

When there is no remainder, integer division is straight-forward but in general there is an ambiguity:

$(+11)\div(+3)=+3$ remainder $+2$ or $+4$ remainder -1

$(+11)\div(-3)=-3$ remainder $+2$ or -4 remainder -1

$(-11)\div(+3)=-3$ remainder -2 or -4 remainder $+1$

$(-11)\div(-3)=+3$ remainder -2 or $+4$ remainder $+1$

(note that in each case the correct dividend is obtained if the product of the divisor and quotient is added to the remainder.) Students presented with this situation are sometimes surprised to find that $(-a)\div(+b)$ is no longer equal $(+a)\div(-b)$ except in the case when there is no remainder. There seems to be little advantage in choosing any particular solution to this problem but discussions with students have suggested four reasonable alternatives:

a) any remainder must take the same sign as the dividend
b) any remainder must take the same sign as the quotient
c) any remainder must take the same sign as the divisor
d) any remainder must be left positive.

In the classroom different groups of students have voted for different alternatives but no-one seems to mind about the final decision since it is difficult to find a problem that makes sensible use of integer division with a negative operand.

The choice of the fourth alternative is rather arbitrary but it is at least consistent with the decision taken over INT and FRAC and it is the solution with the most easy-to-learn rule.

Double length working and overflow. There are three closely related aspects of basic computer science with which the student must be brought face to face. Firstly, there is the question of overflow. Overflow falls into three main categories which are described here under the three headings of floating point overflow, integer overflow and logical overflow. Whilst floating point overflow is never deliberately achieved by good programming, integer overflow might occur either accidently or on purpose and logical overflow (or the losing of bits during a shift instruction) is probably always deliberate.

It has been felt necessary to find a method of protecting the novice from accidental overflow (or at least of providing him with a warning) whilst at the same time allowing the more experienced user to overflow the machine without stopping his program.

Secondly there is the problem that the result of multiplication is, in general, equal in length to the sum of the lengths of the two operands (and of course the nearly converse situation for division[1]). It is probable that teachers will want their students to consider the difficulties associated with double length products and dividends for part of the time but it is possible that some teachers will feel that more successful work can be done with paper and pencil.

Bearing in mind all three of these difficulties the following design for BBC3 has been implemented:

[1] Not quite converse since a quotient is not necessarily single length.

Cell 5 has been reserved as an overflow indicator. It is initially set to zero by the LOGIN sequence and with such a value single length multiplication and division take place so that arithmetic overflow causes an error interrupt and the message INTEGER OVERFLOW to be displayed. However, if the user wishes to ignore the overflow all he needs to do is to ask for his program to 'CONTINUE' (the SCR is stepped back after an error interrupt). If the overflow indicator is positive, no such warning is output, but otherwise the arithmetic is the same. If the overflow indicator takes a negative value, double length working is implemented with accumulator 2 storing the most significant end of a double length product (or dividend), and accumulator 1 the least significant end. Overflow from accumulator 1 is fed into the bottom end of accumulator 2 and overflow from accumulator 2 causes an error interrupt with the message DOUBLE LENGTH OVERFLOW displayed. Both sign bits are correctly set.

In each of these three cases, the overflow indicator is increased by one so that it is always possible (although sometimes a little cumbersome) to test to see if overflow has occurred.

Floating point overflow is a separate issue, does not affect the overflow indicator, and causes FLOATING POINT OVERFLOW to be displayed halting the program, whilst logical overflow causes no special action at all.

Library routines It was felt necessary to point out at a fairly early stage the difference between those functions which these days would almost certainly be implemented by straightforward hardware in a real machine, and those which are more likely to be associated with subroutines. At the same time, students need to be relieved of the tedium of having to write, for instance, routines for READ and PRINT (although these might well provide useful material for examples).

The student is probably encouraged to think of these routines as software packages written in BBC3 machine code and stored outside his area of store. At run time the computer suspends his program when a library instruction is reached and one of these software packages is entered on his behalf. On exit, the computer returns to his program from the point where it left off. The details of the library routines themselves (as opposed to the mechanics of entry and exit) can be left in the hardware/software 'grey' area, but it is more convenient, for the purpose of this description, to regard them as an integral part of the machine.

All these routines (which use the accumulator as operand) are compiled into a LIBR function which has the special property of using the address field exceptionally to specify which particular routine is required. Although it was possible to arrange for library calls to use the address field in the standard way (eg:

```
    LIBR+4
```

could have been compiled into

```
    LIBR 1023
1023 +4          )
```

and although this would have been only a little more artificial than the similar decision taken over the shift instructions, there did seem some point in making an exception here, if only to emphasise the difference between the LIBR routines and the other functions.

Most of the routines provided are fairly standard and require no comment, just as in the case of the functions in the order code, they are modified by the type bits of the operand before they are obeyed.

In particular, PRINT is almost completely four separate routines, one of which is entered according to the operand type bits. The decision to output P-words in mnemonic form has greatly eased the problem of listing a problem and the decision to output the four characters

in an S-word has made the output of strings easy to handle. The printing of an S-word is insufficient to determine the corresponding binary pattern exactly in two respects:

a) If less than four characters are present, there is no way of determining which quarter (or quarters) of the word has been left blank.

b) If an S-word has been used for some special purpose eg., as a collating constant, and has perhaps been input into the computer in octal, there may be non-existent characters whose decimal representation is between 26 and 32.

So that a user can distinguish between these two cases and so that he can decide the exact storage in case a), the teleprinter bell is rung for each occurrence of 'no character' within an S-word during PRINT. F-words are terminated with a single space and I-words with two spaces so that layout is not spoilt if a mixture of F- and I- words are output. Emergency printing takes place if I- or F-words are to be output which are too large in magnitude for the current PLACES setting. This emergency printing takes place on a new line (so that page layout is spoilt) to eight places for I-words and five places (in decimal and exponent form) for F-words.

READ instructions regard data as S-words if the quotes marker is set and distinguish between F- and I-words by the presence or absence of a decimal point (or subscript 10). For the sake of the unified approach, consideration was given to the possibility of P-words appearing among data, but it seemed unlikely that data containing P-words would be of any use in a conversational environment. By allowing input via 8 octal digits, preceded by a P, and I, an F or an S and enclosed in brackets a suitable compromise was reached on this issue.

The presence of an apostrophe during a READ instruction at a time when the quotes marker is not set causes the READ instruction to be cancelled and the computer reverts to a command word state for that user.

Two other functions which require special comment are INT and FRAC. INT finds the integer part of an F-word and leaves an I-word unchanged. The definition is similar to entier in Algol but numbers within a tolerance setting of the next greater integer are rounded up. FRAC always has a positive result so that INT(A) + FRAC(A) is always approximately equal to A.

No immediate operands exist in the computer. The reason for this is the difficulty associated with accommodating all but small positive integers in the address field (or even smaller positive or negative integers). This difficulty would have been tolerated had not an acceptable alternative been available via the software of the assembly language.

The use of literals with JUMP instructions is interesting. A.J.T. Colin in IMDAC takes the view that transfer control to the cell whose address is 72 should be written using an immediate operand since the instruction is to be 'done' with 72 itself and not its content. This point of view has been rejected for the following reasons:

a) It is not only a question of the definition of JUMP. Defined as replace C(SCR) by the content of the cell given in the address part, one must take the IMDAC view, but defined as jump to the instruction which is C (cell given in the address part) then the BBC view is more logical.

b) The editing capabilities given under 'ADVANCE' could not have been provided alongside the IMDAC definition.

c) No detectable confusion has occurred in the mind of a single student on this point using the BBC machine.

d) Presumed modes are associated with each function in IMDAC; this is against the philosophy of the unified approach which is attempted in the BBC machine.

5. DEFINITION OF THE ASSEMBLER

The assembler is a program which accepts as data source code in the form of source program lines. Each source program line corresponds to one main word of object code and translation is strictly on a line-by-line basis. Sometimes the translation of a source program line causes the compilation of a subsidiary word of object code and sometimes it causes a value to be assigned to, or a modification of, an index register.

Syntax

⟨source program line⟩ ::= ⟨location⟩⟨source program word⟩
⟨comment⟩⟨new line⟩

⟨location⟩ ::= ⟨numeric address⟩⟨space⟩

⟨source program word⟩ ::= ⟨S-word⟩|⟨P-word⟩|⟨F-word⟩|
⟨I-word⟩|⟨octal⟩

⟨comment⟩ ::= ⟨no character⟩|⟨space⟩⟨string⟩

Semantics

A location is used to specify where the main word of object code is to be stored and must lie within the bounds 16 to C(8) when the source program word is a P-word. When the assembler is in edit mode, the location is used to fix the position of the main object word. At other times its value is checked against the current value of the assembler storage pointer, if there is a discrepancy, the assembler storage pointer is reset and a warning is output.

Source program words are the main part of the assembly language. They have a number of different variants.

A comment is ignored by the computer, has no affect on the object code and is not kept. It is optional.

Syntax

⟨S-word⟩ ::= ⟨quote⟩⟨actual character⟩⟨character⟩⟨character⟩⟨character⟩⟨unquote⟩

⟨P-word⟩ ::= ⟨take type mnemonic⟩⟨acc⟩⟨general operand⟩ | ⟨put type mnemonic⟩⟨acc⟩⟨address operand⟩ | LDN⟨acc⟩⟨simple address operand⟩:⟨index⟩ | LDR⟨acc⟩⟨const. operand⟩:⟨index⟩ | LDR⟨acc⟩⟨simple address operand⟩:⟨index⟩ | ⟨library mnemonic⟩

⟨F-word⟩ ::= ⟨unsigned F-word⟩ | ⟨signed F-word⟩

⟨I-word⟩ ::= ⟨unsigned integer⟩ | ⟨signed integer⟩

⟨octal⟩ ::=(⟨type designator⟩⟨oct.dig⟩⟨oct.dig⟩⟨oct.dig⟩⟨oct.dig⟩⟨oct.dig⟩⟨oct.dig⟩⟨oct.dig⟩⟨oct.dig⟩)

Semantics

The assembler will accept source program words corresponding to any of the four types. However if the program is to list correctly certain conventions must be followed. In particular, an S-word can only follow a CAPTN routine or another S-word and I- and F-words cannot occur at all except as constant operands. The storage of S-words is left adjusted.

The chief task of the assembler is to translate P-words and the alternatives allowed correspond to those stored instructions that are meaningful and legal. When P-words incorporate literals not previously used, or undeclared identifiers, the translation of a P-word causes the constant or identifier name to be stored in some appropriate cell at the top end of the store. In this case the content of cell 8 is adjusted.

The restrictions imposed on P-words are designed to guard against accidental program errors. However a programmer can avoid these restrictions by inputting instructions in octal form. To a lesser extent octal is useful in allowing non-standard S-words and F-words to be translated and stored and is also available as an alternative for I-word translation.

Syntax

⟨string⟩ ::= ⟨character⟩|⟨character⟩⟨string⟩

⟨character⟩ ::= ⟨no character⟩|⟨actual character⟩

⟨actual character⟩ ::= ⟨alpha character⟩|⟨numeric character⟩|⟨punctuation⟩

⟨alpha character⟩ ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

⟨numeric character⟩ ::= ⟨digit⟩|+|-|⟨subscript 10⟩|.

⟨punctuation⟩ ::= (|⟨quote⟩|⟨unquote⟩|⟨apostrophe⟩|*|/|:|)|=|?|↑|←|≠|;|,|⟨space⟩

⟨digit⟩ ::= ⟨oct.dig⟩|8|9

⟨oct.dig⟩ ::= Ø|1|2|3|4|5|6|7

Semantics

All source code is reduceable to the 59 characters given in the character table on page 57 of the manual. The implementation at Hatfield assumes a one-one correspondence between these characters, the pressing of specific teletype keys on the Hatfield teleprinters and their hard copy representation. Those characters in the syntactic definition that are not equivalent to their Hatfield hard copy representation are given below. Some of these are non-printing, in each case the hard copy representation of the character is enclosed in a pair of periods.

⟨no character⟩	..
⟨quote⟩	.<.
⟨unquote⟩	.>.
⟨apostrophe⟩	.'.
←	.⌣.
⟨space⟩	. .
⟨new line⟩	.
⟨subscript 10⟩	.@.

Syntax

⟨general operand⟩ ::=⟨address operand⟩|⟨const. operand⟩

⟨address operand⟩ ::=⟨simple address operand⟩ |⟨simple address operand⟩:⟨index⟩

⟨const. operand⟩ ::=⟨signed integer⟩|⟨signed F-word⟩|⟨octal⟩|⟨S-word⟩

⟨simple address operand⟩::=⟨space⟩⟨address⟩| *⟨address⟩

⟨address⟩::=⟨identifier⟩|⟨numeric address⟩

⟨identifier⟩::=⟨alpha char.⟩|⟨identifier⟩⟨alpha char.⟩|⟨identifier⟩⟨digit⟩

⟨numeric address⟩::=⟨absolute address⟩|⟨relative address⟩

⟨absolute address⟩::=⟨unsigned integer⟩

⟨relative address⟩::=⟨absolute address⟩+

⟨index⟩::=⟨digit⟩|⟨digit⟩⟨digit⟩

⟨acc⟩::=⟨no character⟩|2

Semantics A general operand causes an address field and an index field to be chosen forthe P-word containing the general operand. Address operands refer to a location either by name or number and const. operands refer to the content of an address. An index field is always given explicitly except in the case of the index zero in which case it can be either explicit or implicit. Its value is an integer from 0 to 15.

Simple address operands correspond to direct addresses when they start with a space and to indirect addresses when they start with an asterisk. When an address is numeric the integer corresponding to it must lie within closed bounds. These bound are at least 0 to 1023 and are more stringent when an address is used as a location. Relative addresses are calculated by adding the content of cell 12 to the integer given and this calculated value is bounded as before.

Identifiers are recognised by their first 3 and last characters (or by all the characters if there are four

or less). A match is sought between the character pattern of the recognised portion of the identifier and the S-words already stored as identifiers at the top end of store. If a match is not found, a new entry is made and in any case an address is calculated by subtracting one from the address of the copy of the identifier at the top of the store. The identifiers themselves are only stored in cells with odd addresses.

Acc specifies the accumulator to be used. Accumulator 2 is given explicitly and accumulator 1 implicity.

Syntax

⟨signed F-word⟩ ::=+⟨unsigned F-word⟩|-⟨unsigned F-word⟩
⟨unsigned F-word⟩::=⟨decimal part⟩|⟨decimal part⟩
⟨exponent part⟩|⟨unsigned integer⟩
⟨exponent part⟩
⟨decimal part⟩ ::=⟨unsigned integer⟩.⟨unsigned integer⟩|
.⟨unsigned integer⟩
⟨exponent part⟩ ::=⟨subscript 10⟩⟨sign⟩⟨digit⟩|
⟨subscript 10⟩⟨sign⟩⟨digit⟩⟨digit⟩
⟨sign⟩::=⟨no character⟩|+|-
⟨signed integer⟩::=+⟨unsigned integer⟩|-⟨unsigned integer⟩
⟨unsigned integer⟩::=⟨digit⟩|⟨digit⟩⟨unsigned integer⟩
⟨type designator⟩::=S|P|F|I

Semantics The integers and floating point numbers whose representations are stored because of the presence of these source program constituents must be representable in the BBC machine. In particular, it is not sufficient that the number corresponding to the final F-word made up from three or less integers should lie within range, so too must each of the integers and sub-expressions from which it is made up. The unsigned F-word 0.0 is not allowed.

Type designators specify the type bits for storage during octal input.

Syntax

<mnemonic> ::=<take type mnemonic>|<put type mnemonic>|
LDN|LDR

<take type mnemonic> ::=<0-15 mnemonic>|<skip mnemonic>

<put type mnemonic>::=X<0-15 mnemonic>|<16-22 mnemonic>|
X<16-22 mnemonic>|<jump mnemonic>

<0-15 mnemonic>::=NTHG|ADD|SUBT|MPLY|DVD|TAKE|NEG|MOD|
CLR|AND|OR|NEQV|NOT|SHFR|CYCR|OPUT

<16-22 mnemonic>::=IPUT|PUT|INCR|DECR|TYPE|CHYP|EXEC

<skip mnemonic>::=SKET|SKAE|SKAN|SKAL|SKAG

<jump mnemonic>::=LIBR|JLIK|JUMP|JEZ|JNZ|JLZ|JGZ|JOI|
SLIK|SNLZ

<library mnemonic>::=SQRT|LN|EXP|READ|PRINT|SIN|COS|TAN|
ARCTAN|STOP|LINE|INT|FRAC|FLOAT|CAPTN

Semantics Each mnemonic defines a six bit field in the instruction word. For the purposes of translation the five function bits and the mode bit are taken together. All 64 possible binary patterns can be compiled for this six bit field with the single exception of 110111 corresponding to XLIBR. This has been left for future extensions.

Library mnemonics cause 010111 to be placed in the six bit function field and the appropriate library number in the address field.

Extra spaces The formal definition of the assembler as given in the preceeding paragraphs has been slightly relaxed in practice. Extra spaces are allowed at places where

a) they would naturally occur and

b) they cause no confusion.

In particular extra spaces may occur

a) between a location and a mnemonic,

b) between the space or asterisk and following address

in a simple address operand and

c) before a location.

Omitted spaces In the same manner the space at the beginning of a comment may be omitted when it causes no confusion. It was found in practice that students wrote

STOP.

in place of

STOP

and this is now allowable. The terminating space of a location can be omitted before a mnemonic but not otherwise. In particular

116+ 27.4 is different from

116 +27.4

The design of the assembler, like that of the machine itself, has been influenced by the desire to teach the subject in a logical order. Thus the space was used as the normal separator between the function and the following direct address because this was to be the first format introduced to the student and it was felt that he would naturally wish to see a space here.

I- or F-word literals can be introduced next and the student quickly sees the need for a distinction in the assembly format. With no immediate operands available, I- and F-word literals were clearly essential but the decision to allow P-words and S-words was harder to make. P-word literals of the recursive type were regarded as an unnecessary luxury at this level so the decision to allow P-word literals in octal form only represents an acceptable compromise. S-word literals are a nuisance since without care the assembler can confuse them with the identifiers which are also stored at the top end of store as S-words. Ensuring that S-word literals are only stored in cells with even addresses and S-word identifiers in cells with odd addresses provides at least a solution that works.

The area at the top end of store which is used automatically needs some but not absolute protection. An attempt to assemble P-words (not in octal) into a location whose address is greater than C(8) probably results from the student being unaware of how far down automatic storage has progressed. He should be and is protected from this type of error. By contrast, the changing of a literal, by re-assigning a new content to the cell used to store the literal, and the consequent effect this would have on the rest of the program is the type of mistake a student must be allowed to make. This type of alteration remains legal but dangerous.

6. THE COMMAND LANGUAGE AND USER FACILITIES

Tel-comp 2[1] provides an integrated interpretive system for on-line computing where the language used to generate a stored program is the same as that used to control the system. Preceding an instruction by a line number causes it to be stored, otherwise it is obeyed immediately. Thus:

DO PART 1

causes the current stored program to run starting from PART 1 while:

2.1 DO PART 1

is stored and causes a jump to PART 1 at run time. The obvious advantages associated with only having to learn one language resulted in serious thought being given to a similar organisation in BBC. It was abandoned only when it became clear that instructions in the stored part of the program were quite inadequate to cope with the control of the system. As it stands now, the command language contrasts sharply with the assembly code; the principle followed is to make them look as different as possible since they cannot in any case be identical.[2] In practice there seems to have been very little confusion and since both command language and assembly code are capable of being introduced via very small subsets, the burden on memory has been small. Once this decision had been taken, it became possible to extend the command language in another direction and to allow the compilation into BBC machine code of instructions being written in a high level language.

[1] Time Sharing Ltd, 142, Great Portland St, London, W1.

[2] c.f. 'The outer and inner syntax of a programming language', M.V. Wilkes, The Computer Journal vol 11 no.3.

'LOGIN'. The logging in sequence serves the following purposes:

a) It enables the user to identify himself and to gain access to any of his three files.

b) It enables the computer to record the statistical summary of his progress. Whilst it is not argued that users should be charged for their time, it is necessary for the teaching staff to be kept in touch with the progress of their students.

c) It fills the user store with zeros except for certain special cells whose contents are set to their initial (presumed) values. It is sometimes argued that the presence of all zeros in a cell renders it no more empty than any other content, and that a user should accept the computer as he finds it. Providing a 'clean sheet' for the user has a number of consequences:

i) Although the responsibility of initialising store locations to zero remains with the programmer, errors in this respect are only detected after a second attempt at running the program; students should, in any case, test their programs with more than one set of data.

ii) It makes the listing of a program a practical possibility; without it, it would not be possible for the computer to distinguish between the current user's program and garbage from the previous user.

iii) While it is dangerous to reply on a clear store for the purpose specified under (i), zero constants which are automatically set are useful in the context of a sparse look-up table with few non-zero values. It is interesting to note that the latest version of Dartmouth Basic as defined by 'BASIC, preliminary 5th edition' from Dartmouth College, sets all variables to zero prior to the execution of a program.

d) When the store has been cleared, the user's name is typed back to confirm the logging in routine and to provide identification for results of programs which are run off line.

The user's name (which follows 'LOGIN') must obey the same rules as other identifiers and must belong to the computer's list of recognised users. The cell where this name is stored is not accessible to the user.

'BREAK'. Students seem surprisingly poor at counting and it soon became evident that it was necessary to protect them from the error of leaving a location blank or using the same location twice. Two methods for doing this were rejected:

a) To allow a student to type in instructions without a location number and for the system to order them for him. Objection: Since labels for jump instructions are not allowed he needs to refer to absolute (or relative) addresses as labels and these need to be included in his image of the program.

b) To arrange for the system to type the next location for him, inviting him to add the instruction. Objection: Eventually, when he comes to the end of a segment of his program, he will wish to insert a command word. One essential, which must be maintained, is for the student to realise that a command word is not stored (and is not therefore preceded by a location number) but is obeyed immediately. If the computer types back the next location number for him, he is likely to be confused if he is now allowed to follow it by something which is not stored.

The method which has been implemented to make sure that a student enters instructions in sequence is to check that the location numbers that he types are in order.

If he breaks the natural order the instruction is still accepted, but a warning message is output and he must use the command 'EDIT' if he wishes to avoid repeated warnings. (This restriction of having to use cells in sequence is relaxed after an error so that a user is free to type immediately the same location number with a correct version of the instruction if he so wishes.)

'BREAK' has two other uses: it specifies the location of the next instruction as a base address for following addresses given in relative form. The C(12) is a base for relative addressing and is reset to the first address typed in after 'BREAK'. This address can itself be relative (using the old C(12) as base). In particular 0+ after 'BREAK' signifies no break in sequence but C(12) is reset. When it is not known in advance (ie when programs are prepared off-line) how much room will be required by compiled statements, relative address form removes from the user the responsibility of anticipating the exact position of the next segment of his program. Relative addresses are translated immediately into absolute form by the assembler. Secondly 'BREAK' is used to reactivate the assembler after initial translation has ceased.

'EDIT' provides the means of inserting instructions out of order so that odd instructions can be corrected without regard to sequence.

'RUN'. Beginners are able to execute their programs from the first instruction they input by typing 'RUN' on a line by itself. The address of this first instruction is stored in cell 10 so that normal entry (which is not necessarily at the instruction with the lowest address) can be automatic. More experienced programmers will want to specify the first instruction and may follow 'RUN' with an address. This provides, amongst other things, the means of storing a number of small programs in the same block of 1024 words.

There is no data store; a program causes DATA? to be output when data is required, and at this point, up to one line of data can be typed in. After END OF PROGRAM has been displayed, or when a program is interrupted, unused data causes the message UNUSED DATA:- to be displayed. followed by the next six characters of unused data. Any such data is lost and does not remain in the line input buffer.

'CONTINUE'. At any time when the keyboard is not active, a process may be interrupted by typing any character. If a running program is interrupted the message INTERRUPTED L= .... is output immediately after the current instruction has been obeyed. By this time the sequence control register has already been incremented so that the C(L) is the instruction about to be obeyed. 'CONTINUE' produces almost the same effect as 'RUN' L. It can be seen that in this way a program may be interrupted, so that amendments can take place or monitoring adjusted and then be allowed to continue.

'TELEP'. The system is used at certain times from teletypes in the computer unit itself. The computer is equipped with two paper tape readers and the operator may specify these (by using the computer keyboard) as input devices in place of the teletype. The user can regain control of his program when reading from paper tape via the interrupt facility or by the appearance of 'TELEP' on the input tape. Remote users (with an adapted teleprinter) can use the teleprinter tape reader in the same way. The command 'PTAPE' effectively causes the computer to press repeatedly the single shot button, and programs that have been prepared off line can be input this way. Input from the teleprinter reader is limited to five characters per second.

'FILE', 'FETCH'. The automatic allocation of space at the top end of the store would make it dangerous to allow partial transfer to or from backing store and

overlaying in particular presents difficulties. A function BSTR was considered in the order code to allow dynamic transfer to or from backing store. Eventually, this function was discarded because of the inadequacy of the Elliott film. The command language provides only for the complete replacement of the user's core store with a copy of backing store of equal size, or the reverse. Three slices of backing store have proved more than adequate for all but the real enthusiast. When a particular student has asked for more than three files, he has been provided with three more via a second user name. The system also provides a number of programs on a read only basis which are called by name. Only brief details of these programs are supplied to the user, on a special file reserved for this purpose, but each contains its own built in documentation: cell 16 always contains an unconditional jump to a part of the program which outputs as a caption its own running instructions, together with any other details which might be required.

An attempt has been made to provide, via this library, programs that the non-specialist can use as tools, programs for demonstration purposes or to illustrate certain programming techniques, and a group of programs which fill in the gaps in a student's basic computer science education left by BBC3 itself. A number of the programs made available in this way have been written by staff and students using BBC3 at the various centres. The following is a selection of the available systems programs.

MONITOR. This is a program which monitors another program by causing it to be obeyed interpretively. It is intended that it should throw some light on how the system monitoring could have been implemented in BBC itself. Instructions are printed out as they are obeyed.

POLISH. Students are invited to type simple assignment statements as data. These are transferred and displayed in both forward and reverse polish form and

are then evaluated from the forward polish form using a recursive technique. Only diadic operators are allowed.

GRAPH. This program uses the teleprinter to draw a graph (or two graphs on the same axes) by typing asterisks (or asterisks and plus signs). Axes are typed as rows of full stops and scales are under the control of the programmer.

JOT. Users are invited to type arithmetic expressions to be evaluated. + - x /↑and = are used together with integers and floating point numbers. The order of precedence is strictly from left to right and no brackets are allowed. Functions may be included by preceding their arguments with the appropriate mnemonic in quotes. The program demonstrates the use of an unusual look-up table with items of different types.

STDEV. This evaluates standard deviation giving either the answer only or the complete solution in tabular form.

FORTUNE 1 and 2. These programs ask personal questions about the user and make appropriate comments on the answers.

OXO. A particularly well written version of a noughts and crosses program which was developed by a schoolboy with only three months programming experience.

PRIB. An alternative print routine which rings the teleprinter bell the appropriate number of times before printing each digit. (BBC3 is used by a group of blind students. The need for PRIB and its implementation arose out of class discussion.)

BINARY. A routine which prints out selected areas of store in binary, separating the various fields according to the type bits.

ASSEMBLER. This is an alternative assembler for the BBC3 machine. It is used to illustrate assembling techniques and to bring home to the student the sometimes forgotten fact that the input of his program requires first another program in store, which treats his program as data.

MIN3. This is the simulation of a three address machine and associated assembler. It is used in a comparative study of machines of different design.

CG319. City and Guild mnemonic code is compiled into BBC3 machine code.

'ADVANCE'. The traditional way to edit a program in an on-line environment is to provide each statement with a line number, and to allow users to edit their programs by inserting intermediate line numbers (usually by using a decimal system so that line 2.1 can be inserted between line 2 and line 3) to label the inserted instructions. The computer then re-orders the source program and compiles again from scratch. A second method (used by Gordon Bull in HSL) is to allow compilation on a line by line basis and to arrange links from one line to the next. This way, the insertion or removal of a line of program is achieved by changing the links and recompilation is not necessary. Similar methods are used for conversational languages which are wholly interpretive.

In an assembly code environment, once the program is in the machine, editing is done by patching, followed up by a retranslation of the corrected program when the logic of the patching has been proved. Retranslation is not possible in an on-line machine code context where a separate copy of the source code is not kept, so that an alternative has had to be found. The use and the limitations of ADVANCE in BBC are best illustrated by example:

incorrect version		corrected version	
100	XCLR SUM	100	XCLR SUM
101	XCLR N	101	XCLR N
102	TAKE N	102	TAKE N
103	SKAE+10	103	SKAE+10
104	JUMP 106	104	JUMP 106
105	JUMP 109	105	JUMP 110
106	READ	106	INCR N
107	XADD SUM	107	READ
108	JUMP 102	108	XADD SUM
109	rest of program	109	JUMP 102
		110	rest of program

Correction here is straightforward and is achieved by the following:

```
'ADVANCE' +1,  106,  130
'EDIT'
106  INCR  N
```

(assuming that this segment of program does not stretch beyond location 130).

It should be noted that address operands (provided that they are not the pseudo-addresses associated with LIBR) are automatically adjusted if they point to a section of program which has moved. ADVANCE can be used to move a segment of program either forwards or backwards and safeguards are incorporated to make sure that the resultant program does not extend outside the range 16 ⇔ C(8). 'ADVANCE' may also be used to eradicate a piece of program no longer required by moving a piece of non-existent program on top of it. The contents of 4, 7 and 10 are altered (if necessary) so that it is possible to move part of a program part way through its running stage.

Difficulties arise when an address is indexed or when an address is referred to indirectly. In general, indirect or indexed jump instructions at the end of a subroutine will present no problem since the link address is planted at runtime, but at other times automatic

relocatability becomes impossible. Consider the following:

```
IPUT CH
TAKE+64
XOR  CH
JUMP*CH
```

which expects an F (for female) or an M (for male) from the data and causes a JUMP to either 70 or 77. Under these circumstances the instructions in 70 or 77 are not relocateable by any standards. Similar situations can arise out of indexed modification, but all are rare. In the vast majority of cases indexed or indirect addressing refers to work space and not to another part of the program, and in practice the need to relocate work space does not arise. Despite its limitations, 'ADVANCE' (which first appeared in BBC 1) has proved to be a very useful editing tool.

'TRACE', 'CHECKA', CHECKN' and 'REMOVE'.
It is important that the user should not find it necessary to plan any monitoring at an early stage. Furthermore, monitoring must provide him with the information he wants and be easy to switch on and off. Finally, the format of any output must be such that it is not easily muddled with the rest of the output, does not waste too much paper or cause output to overwrite itself at the extreme right of the page. Monitoring on all jump and skip instructions has proved a useful quick solution if the problem has gone wrong without giving any indication of where.

'DUMP' provides the facility for part of the store to be output when the program is not running, as opposed to monitoring, which is concerned with dynamically recording store contents during the execution of a program.

'LIST' is similar to 'DUMP' except that a distinction is made between work space and the program itself. The way the assembly language has coped with literals and identi-

fiers has been carefully designed so that declarations can take place at any time, and the computer is still able to distinguish work space from identifiers, both at run time and during a 'LIST'. Basically, the following rules are adhered to:

a) Identifiers are stored as S-words in cells with odd addresses,

b) work space associated with the identifiers is the cell immediately preceding the identifier itself and,

c) literals are stored in odd or even cells except that S-word literals are always stored in even cells. 'LIST' provides additional information in the form of ignorable comments and a complete 'LIST' of a program includes a 'LOGIN' sequence and the necessary 'BREAK' instructions to enable the program to be successfully reinput if necessary.

The ordinary PRINT routine has been used throughout 'CHECKA', 'CHECKN', 'LIST' and 'DUMP' for cell contents, the only modification being that of suppressing the new line before F or I words that are out of range.

It would have been possible to search the identifier directory and constants table with a view to reconstructing the source image of the program completely, instead of using this information merely for comments. The temptation to do this was resisted since it was thought that the implemented alternative was practically as readable and kept the student in close contact with the basic machine code. He could also see clearly the method that had been used to substitute absolute addresses in place of his constants and identifiers.

'DECLARE'. The BBC software tries to steer a compromise between the inflexible necessity of having to declare identifiers before use as in Algol, and the dangers

(mainly associated with mis-spelling) of allowing identifiers without declaration as in some languages.

'ASSIGN'. The one characteristic which designates a language as high level is probably its ability to handle arithmetic expressions without restriction. The command language of BBC allows the user to formulate arithmetic expressions, using any declared identifiers or literals in proper algebraic combination. Any of the diadic operators + - * / ↑ together with monadic + - and library routines are permissable. Subexpressions must be properly bracketed and library routines must contain bracketed arguments. The usual rules governing laws of precedence apply so that operations are performed in the order: bracketed expressions, involution, monadic minus, multiplication and division, addition and subtraction. When equal precedence occurs evaluation is from left to right.

Everything described so far follows the accepted principles for compiler writing but there are a number of special features which need mentioning. The presence of type arithmetic and the absence of type designation to identifiers has meant that the construction of the arithmetic code is simpler than it would have been otherwise. Array work has been excluded largely because of the difficulty of reserving a special area of store for arrays but also because it was not easily possible to protect the user from subscript overflow (a very common mistake for beginners). Library routines provide the usual functions normally associated with this type of arithmetic. A number of these are inappropriate (eg STOP and LINE), but PRINT has turned out to be surprisingly useful:

```
A := PRINT(B + SQRT(C - PRINT(D)))
```

causes D to be printed, followed by the new value of A, as well as the assignment. BBC 1 allowed users to add a function of their own to the library functions, but this has been dropped in BBC3 because it was never very successful in the classroom.

It was tempting to allow users to incorporate the logical functions into arithmetic expressions, but it became too easy to loose track of the type of results obtained this way so that this was not implemented.

Compilation of arithmetic expressions is, of course, into the machine code of the hypothetical computer, not the Elliott. Nevertheless, workspace outside the user's store is set aside for use at compilation time (fixed length since arithmetic expressions cannot be more than one line long). The BBC machine code that is generated is typed back to the student for a number of reasons:

a) It is the only stored image of his program that he possesses and alterations that he may wish to make will probably be made at this level.
b) BBC is designed to break down the high/low level barrier.
c) The user needs to know at least how much code has been generated in case he wishes to revert to low level programming.

The need to show the student the code that has been generated by automatic programming has meant that it must be 'clean'. It does not matter if the code is not as short or as efficient as anything he could generate himself (it will probably do his ego good to discover that he can beat the machine at something), but it should be devoid of apparently pointless redundancy.

Diadic operators are sought according to precedence, and the corresponding operands are found. The first operand is loaded into accumulator 1 with either a TAKE or NEG instruction and subsequent monadic operators are used to generate code. The second operand is loaded into accumulator 2 if monadic operators are present but not otherwise, and finally the diadic operator is used to that the code generated leaves the result in accumulator 1. Temporary store required at this stage (as opposed to compile time

work space) needs to be in the user store. This temporary space is reserved at the top end of the store by automatically declaring the identifiers BBC1 BBC2 BBC3 ..... as required and it is released for future use as it becomes available. These reserved identifiers may not be declared directly or by default, but once declared, they may be used in the ordinary way, so that a user maintains the right to amend his program at assembly code level.

When compilation is complete, but before it is typed back to the user, redundant PUT and TAKE instructions are eliminated. This is followed by the 'LIST' routine which causes the display of an annotated version of the newly compiled segment of program.

The code generated is inferior to that generated by hand in the following respects:

a) No use is made of X-mode,
b) Operands of commutative operators are never exchanged,
c) 0 and 1 are not dealt with exceptionally but address 0 is always substituted for the literal +0,
d) Common subexpressions are evaluated afresh each time they are met,
e) Logarithms and exponentiation are always used for involution.

'LOOPV', 'LOOPN' and 'REPEAT'. The purpose of the loop instructions is twofold. Firstly, it provides high level facilities for controlling loops and secondly, it brings the student face to face with the difficulties (his own and the compiler writer's) associated with call by name and call by value. This problem is famous for its difficulty and every attempt has been made to remove the unessentials so that the central issue is left unclouded. For this reason it was decided to let the fourth parameter represent the number of times through the loop and not

the final value of the counting variable as is the case of a for ... step ... until ... do statement in Algol. No block structure and no procedures are present in BBC so that the equivalent of a for ... do statement became a natural target for this type of analysis[2].

The actual facilities made available to the user are best illustrated by two examples (see appendix 2, Figs. 3 and 4). Both examples correspond roughly to for ANS := D*D+4*C step 2*B-sqrt(D), C/(C-B) times . Here, D*D+4*C, 2*B-sqrt(D) and C/(C-B) can be any arithmetic expressions obeying the same rules as 'ASSIGN' provided that the whole statement will fit onto one line.

If this statement is called by the command 'LOOPV' the code which is compiled (and typed back to the user) is equivalent to the following:

```
ANS := D*D+4*C
STEPSIZE := 2*B-SQRT(D)
TIMES := C/(C-B)
A fixed routine (with variable addresses).
```

The fixed routine organises the incrementation of ANS and arranges for the loop to be repeated, in this case INT(C/(C-B)) times. The number of times through the loop and the step size are constant; both are dependent on the values of on entry of the variables B, C and D.

When the same statement is compiled under the control of 'LOOPN', the code is generated differently so that the step size and number of times through the loop are evaluated afresh at each incrementation.

It is hoped that students will be encouraged to study the two versions in a number of different situations, so that they can appreciate the advantages and disadvantages of the general concepts of call by name and call by value.

[2]Especially so since the attention paid to this topic in Algol 68.

A limitation onthe system is that it is not possible to call one of the two variables (step size and times) by value and the other by name, but it must be remembered that BBC is concerned with concepts rather than details. On the other hand the ability to nest loop instructions is fundamental and in this context depends on the ability of the compiling routines to seek fresh workspace on each nested entry. The automatically declared identifiers are reserved uniquely by the insertion of an S-word (because an S-word cannot be garbage from a previous attempt at running) corresponding to the name of the counting variable. This S-word remains there while the intermediate instructions are generated and is only cleared by the command 'REPEAT'.

It is easily possible to recognise a 'REPEAT' command and to associate it with a particular loop, without the need to add the counting variable as parameter. Nevertheless it was decided to ask for the parameter as a check since failure to nest this type of statement correctly is a common mistake. A second reason for its inclusion was to provide a visually more complete document. The parameter itself is used to store three pieces of information about the loop:

a) The address of the last automatically compiled instruction (JEZ), so that the instruction can be modified to serve as a true exit to the loop.
b) The return address so that the unconditional JUMP can be compiled to close the loop.
c) A bit describing the type of loop which is being closed.

The information given under c) is required so that the correct number of reserved identifiers can be freed (2 for 'LOOPV' and one for 'LOOPN').

It has been found that one effective method of demonstrating the advantages and limitations of the two types of loop instruction is to arrange a demonstration using two terminals side by side. At one terminal a program is developed using 'LOOPV' and at the other a program is

developed using 'LOOPN', but otherwise identical to the first program. Examples are chosen with long expressions for both STEPSIZE and TIMES, with no changes made to any of the variables within STEPSIZE or TIMES and with a PRINT instruction (of the counting variable perhaps) within the loop itself. These programs are run together so that a 'race' is initiated and students can appreciate the considerable time saving associated with 'LOOPV'.

The same programs can now be modified to show that when parameters used within STEPSIZE and TIMES are altered in the loop itself, no change is made to the output from the program using 'LOOPV' but that changes do result in the output from 'LOOPN'.

'MESSAGE'. Students like to be able to lay out their results with proper headings and it is felt that they should be encouraged to do so. The library routines READ, PRINT and CAPTN have made it easy to handle groups of characters but it does become a little tedious to split messages into groups of four letters. The command 'MESSAGE' does this automatically for the student so that:
'MESSAGE' <MARY HAD A LITTLE LAMB> causes:

```
93 CAPTN
94 <MARY>
95 < HAD>
96 < A L>
97 <ITTL>
98 <E LA>
99 <MB>
```

to be compiled onto the end of the existing program. As with 'ASSIGN', 'LOOPN', 'LOOPV' and 'REPEAT' the compiled code is typed back to the student.

7. IMPLEMENTATION

The implementation of BBC on one of the Elliott 803's at The Hatfield Polytechnic has placed severe demands on existing hardware and has necessitated the construction of a specially designed multiplexer to provide the interface between the computer and the three teletype consoles.

As it has been necessary to limit the total budget on the project, the facilities described elsewhere in this thesis are less than those that could have been included if computer speed, storage size and sophistication had allowed. Possible refinements that could have been included in a more comprehensive environment are detailed in chapter 9. Apart from these refinements, the system as implemented is identical to the one described in the manual and it is this implementation which is described. Although the whole system is written in Aberdeen SAP (as assembler for the 803) the various routines described are given either in flowchart form or are translated into Algol for easy reading.

Hardware limitations The existing configuration consisted of an Elliott 803B computer with floating point hardware, a set of three 'Victorian' film handlers and an incremental digital plotter. In May 1967 a simple interface was added to this configuration to allow the implementation of HSL. (HSL is a dialect of Basic which has been implemented by Gordon Bull at Hatfield). This interface required the dedication of the computer to a single on-line user, using a continuous scan technique when the input of characters was expected. A very simple interrupt was provided which could be triggered by the typing of a key when input was not expected, but it was never possible to use this to catch individual characters

being input since by the time the interrupt was triggered the incoming character was already lost.

Despite the inadequacy of this device, it was used extensively from May 1967 by BBC 1 and from September 1968 by a single user version of BBC3 (known as BBC2) until the multiplexer was delivered in March 1969.

The digital plotter has not been considered an appropriate output device for BBC and has not been used.

The magnetic film (originally developed by Elliotts for the 405) is slow and inflexible, but it has provided the necessary backing store for students to use in an on-line environment for complete programs. In practice, it spends much of its time searching and although computing can continue while a single search is being made, the system would become film bound if dynamic transfer of information to or from backing store was allowed.

The Elliott floating point hardware is relatively fast (addition takes $1\frac{1}{2}$ times as long as fixed point addition), but floating point overflow on the Elliott is not under the control of the program. However, because of the smaller exponent in a BBC floating point word, the only floating point computation in the BBC machine (apart from division by zero) which could cause overflow in the Elliott machine is $(-1x2^{127}) \div (-1x2^{-128})$. The incidence of this case must be extremely rare for although there is no check on it, there has never been an occasion when it has occurred.

The flow diagram given in Appendix 2 fig.5 indicates the way in which an Elliott floating point operand is prepared and how the result is condensed at a later stage to a BBC word.

Timesharing on the 803 The decision to base the timesharing aspect of the project on a continuous scan technique rather than using input/output interrupts was made largely because of the interpretive nature of the whole system. As the only code that is obeyed is permanent code, it was very easy to incorporate the necessary scans to test requirements for the input or output of characters from or to the terminals. These scans have to be made at one tenth second intervals at least (the teletypes work at a speed of 10 characters per second). This means that the scan instructions have to be inserted so that, at run time, a scan is made after at least 200 instructions. In the event it was possible to insert them more often than this as there always seemed to be strategic points in the program that were entered far more frequently. It was usually possible to insert them at a time when nothing of value was in the accumulater so that it was normally not necessary to store away the content of the accumulator before the test was made. No modification was made to the original number of scan instructions inserted.

The philosophy behind the multiplexer instructions is best understood in terms of the set of Algol procedures given in appendix 3 and the flowchart given in fig. 1, appendix 2. Using the Algol procedures the scan instructions appear as overleaf.

```
procedure scan;
if mpxrts then
begin integer user,ch;
      for user:=1 step 1 until 3 do if test(user) then
      begin if inmode(user) then
            begin trans(user,ch);  lookup(ch);
            comment characters have to be changed to their
            internal value given by the character code on
            page 57 of the manual;
                  if Irtflg[user]≠0 then Irtflg[user]:=-1
            else if ch=63 then inpointer[user]:=inpointer[user]-1
            else if ch=62 then
                  begin chmode(user,1);  Irtflg[user]:=1;
                        input buffer[user,inpointer[user]+1]:=62;
                        input buffer[user,inpointer[user]+2]:=0;
                  end of complete line,set flag for service
            else if inpointer[user] > 67 then
                  begin inpointer[user]:=0;  error flag:=1;
                        chmode(user,1);
                  end too many characters to a line
            else begin inpointer[user]:= inpointer[user]+1;
                        input buffer[user,inpointer[user]]:=ch
                  end of normal character input
            end of input
       else begin outpointer[user]:= outpointer[user]+1;
                  ch:=output buffer[user,outpointer[user]];
                  if ch=0 then
                  begin inpointer:=0;
                        chmode(user,0)
                  end of complete line of output
            else begin delookup(ch);
                        comment change character to its ASCII value;
                        trans(user,ch);
                  end of output of normal character
             end of output
             end of loop;  scan;
      end a recursive entry is made here to ensure that when the
          routine is finally left no more servicing is already
          required.  This way, the highest priority is given to
          the instructions within scan.
end of scan;
```

The permanent code into which the scan instructions are embedded is made up of the following routines:

a) an assembler
b) a command language translater
c) routines for carrying out the various commands
d) a routine for the servicing of errors
e) an interpreter
f) a set of LIBR routines
g) scan itself.

The computer divides up its time at this second level of priority by doing a single 'job' for each user. This task must be one of the following:

a) translate (completely) one line of program using the assembler
b) translate a command and either obey it if it is a short command, or initiate it if it is a long command
c) continue with a long command already initiated
d) obey a single instruction or LIBR routine for a user
e) do nothing (a user might be in the middle of a line of input or in the middle of receiving a line of output).

Long commands, those that cannot be completed as a single job involve either a film search or more than a single line of output. The 'ASSIGN' command, for instance, is followed by a listing of the new compiled code. This is loaded into the output buffer a line at a time, so that the full task to be undertaken on behalf of a user in this case is a short piece of compilation, taking a second or two of time, followed by a listing which might take as long as a minute. This full task is organised as an initial job of compilation followed by subsequent jobs which will involve reloading the output buffer if it is ready for reloading and doing nothing otherwise.

When reading the following ALGOL summary of this second level of priority, it should be borne in mind that there are a number of global variables reserved for each user which are needed to record the current state for a particular user. One of the most important of these is NJN (next job number). Immediately before a job for a particular user is terminated the correct value of his NJN is selected so that the computer will be able to carry on for him in the appropriate manner when he gets his next slice of time. Many of the other global variables are not shown here.

```
begin integer user; integer array Irtflg,NJN[1:3];
      Boolean array searching,output complete[1:3];
      switch next job:= instruction,more data,command,execute,
             film1,film2,output,Round robin;
      for user:= 1 step 1 until 3 do initial state(user);
      comment sets pointers and flags to their initial value;
Round robin:  if user =3 then user:=1 else user:= user+1;
              goto next job[NJN[user]];
instruction:  if Irtflg[user] ≠0 then begin assemble(user);
                                           Irtflg[user] :=0
                                           end;
              goto Round robin;
more data:    if Irtflg[user]≠0 then NJN[user]:=4;
              goto Round robin;
command:      if Irtflg[user]≠0 then obey command(user);
              goto Round robin;
execute:      interpret(user);  goto Round robin;
film1:;film2: if searching[user] then goto Round robin;
              if NJN[user]=5 then file(user) else fetch(user);
              NJN[user]:=3;  Irtflg[user]:=0;
              goto Round robin;
output:       if output complete[user] then terminate(user)
                                       else list(user);
              goto Round robin
end;
```

Apart from the time taken to execute long programs the response time has been easily adequate. It should be remembered that when an average response time of one second is given this excludes the time taken for execution. It does include the very quick response, given on a line by line basis for the normal translation of instructions by the assembler. The times given below for the various user-jobs are approximate:

'LOGIN' sequence	4 sec
'ADVANCE'	3 sec
'FILE' or 'FETCH' search	up to 1min (but does not hold up other users)
Execute an instruction	1/10 sec
Translate an instruction	½ sec
Compile a full line of a high level statement	3 sec
Load a line of output into the output buffer	½ sec

Practically all 8k of store is used by either the system or the users. (At the time of writing there are only two spare words of core.) It has been necessary, therefore, to write part of the system so that it overlays one of the user areas. The version of the complete system is brought down into the main store with a trigger tape from magnetic film and this version contains all the extra parts needed initially and finally, but not while the system is actually running. Three sets of master names are read in (not necessarily three different sets), the system is initialised (all in one operation), finally a routine is triggered which clears the user areas and the system becomes operational. At the end of the session the overlayed portion is re-entered from paper tape and the error summary output is triggered. This output is one, two or three summaries, according to whether one, two or three different sets of master names were initially entered. Finally, the system is left in a state ready for the re-entry of new sets of master names.

8. BBC3 IN USE

BBC3 fills a need in the area but the way it is being used in courses varies continuously. At the time of writing (February 1970) it is the only on-line system available to outside users and competes internally with three GEIS terminals. When the Polytechnic's new PDP-10 is delivered, it is intended to integrate existing terminals with those for the new system (ref: appendix 2, fig 6), so that for a time users will be able to dial into either computer. How long this period of transition will last will depend on the speed with which the Computer Centre is able to offer an efficient service on the PDP 10, and how long it takes to rewrite BBC for the new computer. Eventually the Centre intend to withdraw the 803 from general use but this will not be before an alternative service is available on the PDP-10.

Currently six external terminals (all teletype 33's) are operational. Five use the GPO datel 200 service and the sixth uses a land line. Three of these have been regularly on-line for scheduled times for nearly a year and the other three started regular connections in October, 1969. The chief internal load is currently for a teachers' course. A total of 18 computer hours per week or 54 terminal hours per week is scheduled BBC3 time, 500 manuals have been issued by the Polytechnic, and the system has been demonstrated in educational establishments in Hampshire, Doncaster, Bedfordshire and Lincolnshire, as well as at a number of schools and colleges within Hertfordshire.

Its wide use has had two opposing consequences. The extensive feedback from teachers in the field has confirmed its educational merit and has suggested many improvements on the one hand but has prevented their implementation on the other. Users need a stable environment so that it

has been necessary to refrain from the implementation of any major improvement at least for the time being.

The six educational establishments with remote terminals include two mixed grammar schools, two colleges of further education, a college of technology and a special school for blind girls of grammar school ability. One of the grammar schools and both the FE colleges offer 'A' level courses in Computer Science and the college of technology runs an HND in Mathematics, Statistics and Computing. In addition a considerable amount of non-examination work is undertaken. The schools use much of their terminal time with 13 and 14 year old children, and courses for 11 year olds and parents have been successfully held.

No formal validation has taken place, indeed it is known that some of the work done using BBC3 was most unsuited to the system but it was used because it was all that was available.

Teachers of Computer Science are liable to lose track of their students' progress when they are using an on-line terminal, especially a system with helpful diagnostics. When the student does come for help it is probably quicker and educationally more sound in many cases to suggest the debugging techniques he should use rather than to provide more positive help. The current implementation of BBC3 attempts to keep the teacher in touch with the practical work by providing statistical information about each potential user.

This statistical summary of the progress of each class is dumped off-line at the end of each session and results are sent to the school or college once each week. The following statistics are provided for each user:

a) times logged in
b) number of instructions compiled (including those set up by automatic programming)

c) number of times END OF PROGRAM has been displayed
d) number of times a program has run so that results have been output
e) number of errors (excluding warnings).

The Computer work in the school for the blind presents special problems. Line-by-line compilation keeps the girls in close contact with the computer, but although all the girls can type so that input presents no problem, output is not immediately available. Off line transcription to braille uses program conversion to a sequence of full stops and spaces on a flexowritier but this is by a postal service with consequent delays. However, some immediate responses are available using sequences of printing, non-printing and bell characters. The students use modification software which incorporates such sequences in error and system messages together with the special subroutine available from file for printing 'audible numbers'. These steps, taken to make on-line computing for the blind realistic are regarded as expedient rather than ideal. On-line braille output looks promising for the future but the existing solution does show that BBC3 is sufficiently versatile to cope with an unusual situation.

9. BBC-10, A MULTI-PROGRAMMING COMPUTER

The BBC3 machine as the student first sees it contains a number of features that are not explained:

a) implementation of the command language and assembler
b) implementation of the library routines
c) the user interrupt facility
d) input and output buffers
e) run time error interrupt
f) the multiprogramming aspect of the system.

In the first few weeks the beginner has enough to worry about and does not usually think too deeply about any of the above points. Probably he should be encouraged to think that they are implemented in some way within the hypothetical computer but not to consider the precise details.

After about a term and a half (containing, say, one hour of theory and one hour of practical per week), the discerning student begins to think more deeply about such concepts as the implementation of the library routine STOP. He discovers that the BBC (as it has been explained to him) is inadequate to cope with every situation. About a term later, the average student will start to raise the same issues. The answers that the students expect are at the flow diagram level, they (the students) are not concerned, and should not at too early a stage be encouraged to concern themselves, with the detailed coding; at the same time they must be seen to be codeable. Later, more advanced students might wish to use the type of facility associated with any extended machine.

BBC-10 The Hatfield Polytechnic Computer Centre intend that an extended form of the BBC will be implemented on the PDP-10 configuration to be installed later this year.[1]

[1] Delivery date for the PDP-10 is June 1970 and for the implementation of BBC-10 is January 1971.

Users of BBC3 have asked that it should be available on the new system and have suggested a number of improvements. Where such improvements have general support and where it is possible to implement them without removing any of the basic initial simplicity these have been agreed. At the same time every effort has been made to ensure that the existing machine is disturbed as little as possible.

The major modifications planned for BBC-10 are outlined below, together with some of the reasons for the changes.

Modification Minor changes have been made to the distribution of bits within the instruction word.

Reason The new instruction format fits better into four six-bit bytes and changes were necessary in any case to allow for the other modifications in the design.

Modification More than one block is available for the user program. Transfer of control from one block to another is possible using the instruction XLIBR (renamed BLOCK) and hitherto unused. A special register within each block called the next block number cell stores the block number (which cannot be negative or zero) to be jumped to, and the address field is used to specify the initial value of the SCR. The next block number cell of this newly entered block is automatically loaded with the old block number.

Reason The basic store size makes this modification necessary for the more ambitious student and, in any case, encourages users to segment long programs into smaller manageable pieces which they can develop separately. Students can now program using simple overlay techniques.

Modification Other blocks are available as data areas. For each user, blocks are numbered from 0 upwards. Block 0 is special and block 1 is the block automatically made available to a user when he logs in. Other blocks must be loaded before they are used. They can be filled initially with zeros or they can be filled with a copy of a file.

The store operand in an instruction normally comes from the same block as the instruction itself, but a special cell within each block, called the operand block number cell (OBN), stores the block number of the block from which operands used in conjunction with a third accumulator are accessed. When the C(OBN) is equal to the current block number or takes a negative value all operands are from the current block. Initially C(OBN) = 0.

Associated with this feature is a library routine called BLTR which transfers blocks to and from backing store, but which cannot be used to load blocks. BLTR has three parameters which are placed in the following three cells. The first is an integer and specifies the direction of transfer (a negative value specifying to, and a positive or zero value from, backing store), and the two cells immediately after contain either as integers the block number to be changed and the file number of the associated block on backing store, or as S-words, the names of the same two blocks. BLTR is a copying routine and the SCR is increased by four instead of the usual one. The LIST routine lists the three parameters following BLTR.

Reason Elementary data processing examples now become more realistic and programs using large arrays can be written.

It should be pointed out that BLTR is a library routine and a hardware instruction for movement to and from backing store is still missing even from the extended machine.

One expected difficulty on the PDP-10 is the problem associated with fast retrieval from backing store. An advantage that the 803 implementation has is a true feeling for the essential difference between main store and backing store. This is because the magnetic film is slow so that students using the system locally are

able to initiate a search, and then see the film turning in response to their request. (The 803 generally is slow enough to allow a number of routines to be understood in this direct way.) By contrast PDP-10 retrieval (from disc) is likely to appear instantaneous.

In an attempt to overcome this problem, it is planned that users of BBC-10 will have to specify, before they run a program, which blocks are to be loaded and what their initial content is to be.

Modification The user now has the option of servicing his own run time errors. These errors still cause an interrupt to one of a set of registers within the supervisor, but the supervisor returns control to the user if a corresponding register in block 0 is non-negative. If the content of this cell is positive, this specifies an address within the current block to be jumped to, and it is the programmer's responsibility to ensure that the error is properly serviced from here. From the user's point of view this jump is equivalent to JLIK so that the user must use the link address to find where the error occurred. If the content of a user's error register is zero, the corresponding error interrupts are effectively disenabled completely so that most illegal instructions are equivalent to NTHG.

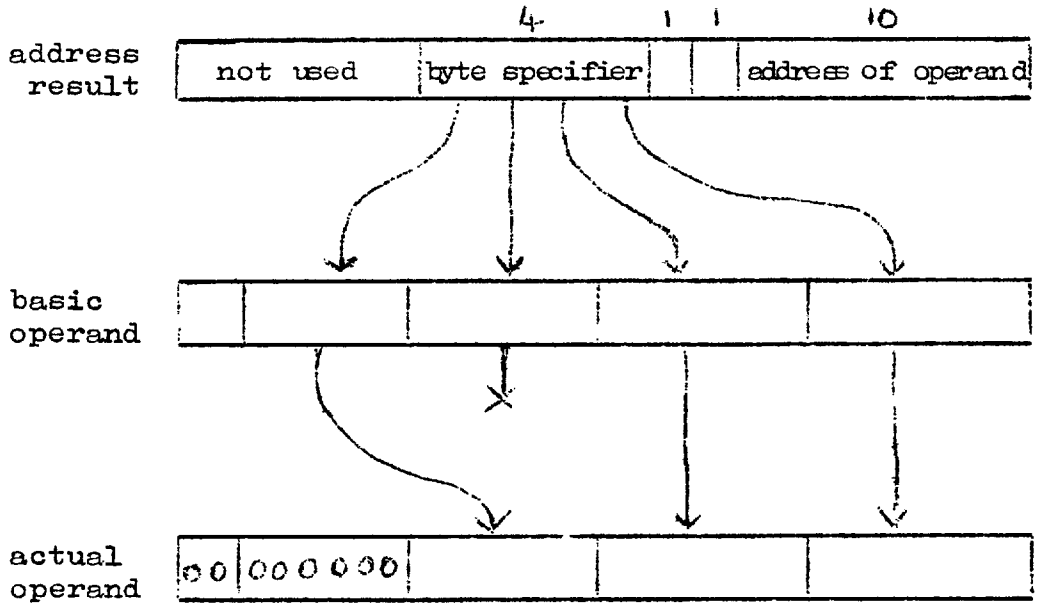
There are certain errors (eg., 'No instruction in this cell') which it is dangerous to treat in this way, but the user is still given the freedom to service these errors if he so wishes.

Reason Users writing their own software have specially asked for this facility. In particular, the implementation of C and G 319 mnemonic code in BBC3 contains limitations in this area.

Modification In the past the calculation of the address of the operand within the instruction execution cycle has been

concerned with a 10 bit result. When the address is direct and unindexed, this is all that is possible, but in general a full word is available and within the extended machine this full word, called the address result, has been put to use.

After the address calculation is complete the new machine examines bits 11 and 10 of the address result for special action. Bit 11=1 designates monitoring while bit 10=1 specifies that the function must operate in byte mode. Under this mode, the store operand is first calculated from the address result according to the following rule:



The bits 0 to 9 access store in the usual way and the result is shown in the diagram as the basic operand. At this stage bits 12 to 15 of the address result are examined, to see which bytes from the basic operand are to be copied across to the actual operand. A one in the byte specifier (shown in the diagram) indicates that a byte is to be copied across and a zero indicates that it is to be ignored. (Thus in the example given the byte specifier must be 1011.) All the bytes that are to be included in the actual operand are concatenated and right adjusted with any empty part filled with zeros.

The actual operand is always an I-word. The function is then obeyed in the usual way. Finally, any result that is to be loaded back into store is first loaded into the actual operand cell and the process is reversed. It should be noted that bytes within the store operand which are not accessed remain unchanged and that the type of the store operand (which is unused) also remains unchanged.

Associated with this extension is a new function called IBP (increase byte pointer). This function assumes that the store operand is a byte pointer used with an array of single bytes. When obeyed a cyclic right shift of one is performed on bits 12 to 15. As a separate operation the address stored in bits 0 to 9 is incremented if the cycle shift caused a 1 to be moved from bit 12 to bit 15, but not otherwise.

The following binary patterns illustrate the result of successive IBP instructions.

0 —— 0	0010	01	1111111110
0 ——0	0001	01	1111111110
0 —— 0	1000	01	1111111111
0 ——0	0100	01	1111111111
0 —— 0	0010	01	1111111111
0 —— 0	0001	01	1111111111
0 —— 0	1000	01	0000000000
0 —— 0	0100	01	0000000000

Reason Byte manipulation is a concept which belongs to a study of basic computer architecture and a machine of this complexity is certainly likely to have such special facilities. The cell splits conveniently into four six-bit bytes and the byte instructions are useful to the student who wishes to manipulate his buffer area directly.

Consideration was given to the concept of one or two special byte instructions, but the proposed design gives the user a convenient method of producing a temporary I-word operand. By modifying its address with an index register,

whose content is (I00172000), this effectively turns an ordinary instruction into a byte instruction with all bytes present.

Modification Block 0 is a data block with special properties. It contains individual cells that record for the user information about his files, together with a number of constants that take presumed values, such as the number of digits to be output during a PRINT instruction. The input and output buffer areas are also within this block. They have been extended in size, and data to the user program uses a separate input area from the rest of the input.

Special cells within block 0.

GIP1. General input pointer no.1. This cell points to the next character to be accessed from the general input buffer.

GIP2. General input pointer no.2. This cell points to the next available space for a character within the general input buffer.

DIP1. Data input pointer no.1. As GIP1 and GIP2 but for

DIP2. Data input pointer no.2. the data input buffer.

OP1. Output pointer no.1. This cell points to the next available space for a character to be placed for output.

OP2. Output pointer no.2. This cell points to the last character that was output.

All these pointers use the new byte format so that if the user wishes, he can use Accumulator 3 and one of these pointers to access or load characters into the buffer areas by normal programming, rather than by relying on IPUT and OPUT. For each buffer a pair of pointers is required. One records the byte position between the teletype and the buffer and the other records the byte position between the buffer and the program. When these pointers are pointing to the same character position the corresponding buffer is empty.

OFLO. Integer overflow indicator This was cell 5 in BBC3.

TOL. Tolerance setting This is used with conditional jump instructions that have F-word operands. In BBC3 it was cell 6.

CLOCK. Described elsewhere in this chapter.

CBN. Current block number. This cell records for the supervisor the currently active block. It is not expected that the user will find a need to access or change this.

STP. Stack pointer Block 0 contains an array of 64 cells used in conjunction with the special stack register within each block described elsewhere in this chapter. STP is used to point to the top of this stack.

STAT. User status cell. Bits within this cell record the current state of the user. The following list of user attributes is associated with particular bits within STAT:

a) Records whether a user is in 'system' or 'user' status. In user status his program is either running or waiting to run but in system status, he is making use of the systems software.
b) Input/output is, or is not, in progress.
c) User is in an input or output state.
d) A running program requires or does not require a user initiated interrupt.
e) A user is, or is not, waiting for a transfer from backing store.
f) A user is, or is not, waiting for a transfer to backing store.

MONR. Monitor status In BBC-10, only one bit is used within the instruction word to indicate when monitoring is required. Different types of monitoring are given by the 3 least significant bits of MONR, so that the user has a full range of monitoring alternatives.

BASE. Base address This replaces cell 12 in BBC3. Relative addressing uses C(BASE) as the base address.

QUOTE. The quotes marker This replaces cell 11 in BBC3. There is only one quote/unquote character and its occurence within data (whatever the context) changes C(QUOTE) from 1 to 0 or from 0 to 1.

PLACES. Cell 9 in BBC3.

ASP. Assembly storage pointer This is normally used to check that a user is inputting instructions into consecutive cells. It is used by the automatic programming routines and is negative in EDIT mode.

The rest of block 0 contains:

a) 14 cells which are used to store the names of up to 7 loaded blocks and 7 blocks on file.

b) About 10 cells which are used as pointers for run time error interrupts. Initially they are all set to -1 to indicate that run time errors are the responsibility of the supervisor.

c) About 6 cells which point to the start of various system messages which are not associated with errors.

d) The messages themselves.

e) 64 cells used for the stack.

f) Three buffer areas.

Reason In BBC3 the cells with presumed settings have been brought to the attention of the student at too early a stage in the course. In the manual 'LIST' is introduced on page 8 and at this stage the user is sometimes bewildered to find that constants have crept into store without him being aware of it.

Each block still contains a number of cells which record properties of that block but initially all the associated constants start at zero. Special cells within each block are now reduced to the following:

Address	Name	Special Property
0	ZERO	C(0) is always zero
1	AC1	The assumed accumulator
2	AC2	The second accumulator. AC1 and AC2 make 1 double length accumulator for double length MPLY and DVD.
3	AC3	Used to access data from different blocks
4	SCR	The sequence control register
5	OBN	Specifies the block to be used in conjunction with AC3
6	REM	Contains the remainder after integer division
7	LINK	Stores the link address after JLIK
8	STACK	The access point to the push-down stack
9	NBN	Next block number. Used by the supervisor to transfer control from one block to another
10	FIRST	Stores the first address of the user program
11	PROT	The protection cell. Records the number of cells reserved at the top end of store for identifiers and literals
12	NAME	Contains as an S-word the name of the block

Modification A place in the order code has been found for a new function called POWR. POWR N means a':= a↑n. The type of a' is shown by the following examples in which the presence of a decimal point is intended to indicate an F-word.

$2^{3} = 8$ $2^{-3} = 0.125$ [1] $(-2)^{3} = -8$ $(-2)^{-3} = -0.125$	$2.0^{3} = 8.0$ $2.0^{-3} = 0.125$ $(-2.0)^{3} = -8.0$ $(-2.0)^{-3} = -0.125$
$2^{3.0} = 8.0$ $2^{-3.0} = 0.125$ $(-2)^{3.0}$ undefined $(-2)^{-3.0}$ undefined	$2.0^{3.0} = 8.0$ $2.0^{-3.0} = 0.125$ $(-2.0)^{3.0}$ undefined $(-2.0)^{-3.0}$ undefined
$2^{0} = 1$ $(-2)^{0} = 1$	$2.0^{0} = 1$ [2] $(-2.0)^{0} = 1$

[1] A purist might demand the answer 0 remainder 1.

[2] Perhaps 1.0 would be more consistent. The difficulty arises because there is not a one-one correspondence between real numbers and F-words. If 'a' represents a real number

$$a^{0} = \frac{a}{a} = 1$$

but if 'a' represents that subset of real numbers that are nearer to the F-word representation of the real '$\bar{a}$' than they are to any other F-word representation then

$\frac{a}{a} = 1.0$ appears more logical.

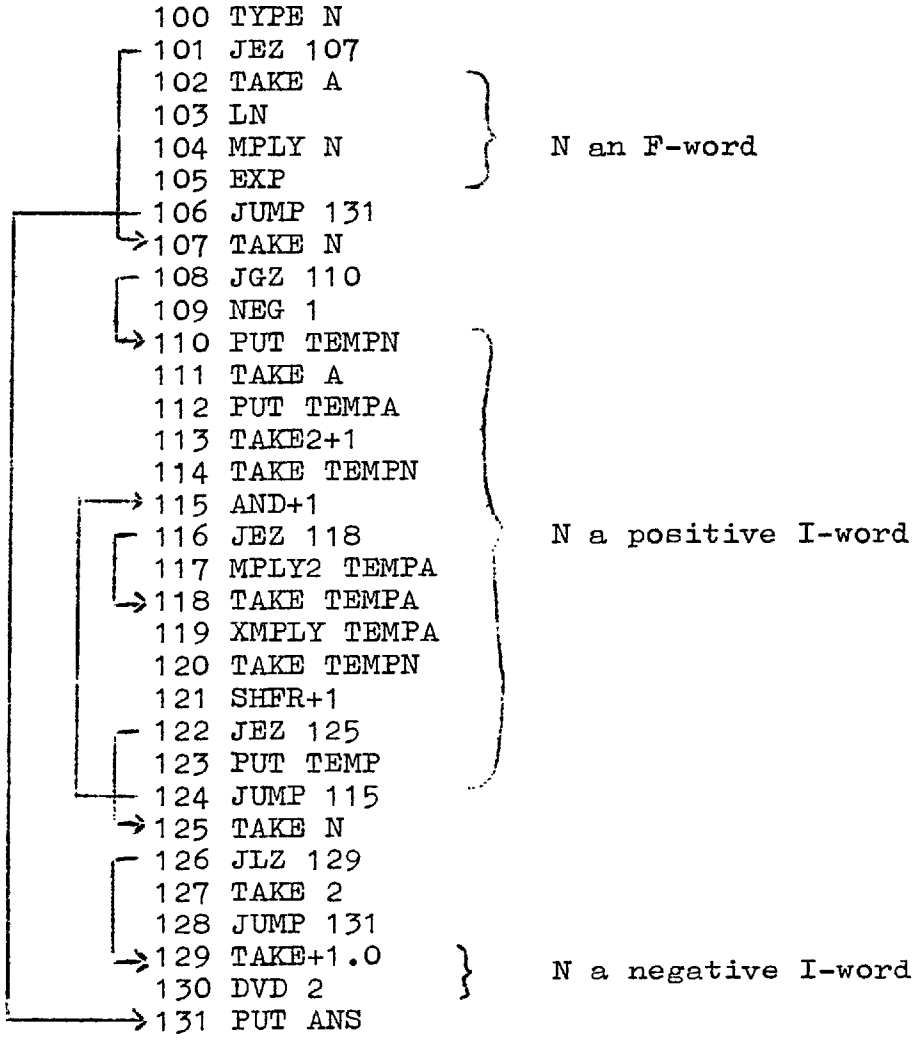
Reason These results are already well-known and it is usual for compiler designers and writers to follow the spirit of them if not the letter. Because type is allocated dynamically the BBC machine and associated software have special problems in this direction.

As it stands, 'ASSIGN' ANS:=A↑N is compiled into:

```
TAKE A
LN
MPLY N
EXP
PUT ANS
```

so that the result is an F-word when A is positive and is undefined otherwise.

Compilation could have resulted in:



but no one would seriously suggest such a solution.

The function POWR solves all these problems so that compilation becomes

```
TAKE A
POWR N
PUT ANS
```

Modification C(9) has always been used by PRINT to specify the number of places for I- and F- words. A negative value for the equivalent of cell 9 in the new machine, is now used to specify octal output. This variant is also used during LIST to output P-words used as literals. The format is the same as for octal input.

Reason Apart from the obvious advantages of octal output, the decision to list P-word literals in octal enables such programs to be successfully re-input from paper tape.

Modification A special index register specifies the address of the top of a push-down stack. The function LDR uses this register exceptionally. LDR assigns a new content to the index register but only after the old value has been 'pushed down'. Later, when the top item is removed, the previous value 'pops' back up again. Any function using this register in the address field (directly or indirectly) removes the top item from the stack after use. RUN initialises the stack with all zero values. Although each block has a register with this special property there is only one stack and access to it uses the special address from any block. When the stack has been exhausted, it appears to be the source of I-word zeros and when it is overfilled an error interrupt is caused.

Reason Recursion remains one of the few aspects of computer science which is incapable of much simplification. Factorial(n) remains the easiest example, but the concept is a tricky one and is probably beyond many schoolchildren.

A program, to evaluate factorial(n) recursively, in BBC3, is now given.

```
16 TAKE+200
17 PUT STACK
18 READ
19 PUT N
20 JLIK 100 ------------------> 100 LDR*STACK:7
                                101 INCR STACK
                           ...  102 JEZ 110
                                103 PUT*STACK
                                104 INCR STACK
                                105 SUBT+1
                           ---  106 JLIK 100
                           ---> 107 DECR2 STACK
                                108 MPLY*STACK
                           ---  109 JUMP 111
                           --> 110 TAKE+1
                           ---> 111 DECR2 STACK
                                112 TAKE2*STACK
21 PUT FACN <------------------- 113 JUMP*2
22 CAPTN
23 <FACT>
24 <ORIA>
25 <L(>
26 TAKE N
27 PRINT
28 CAPTN
29 <)=>
30 TAKE FACN
31 PRINT
32 STOP
```

The essential nesting features of the subroutine entries can be shown by running the program with monitoring.

'CHECKA' 100,113

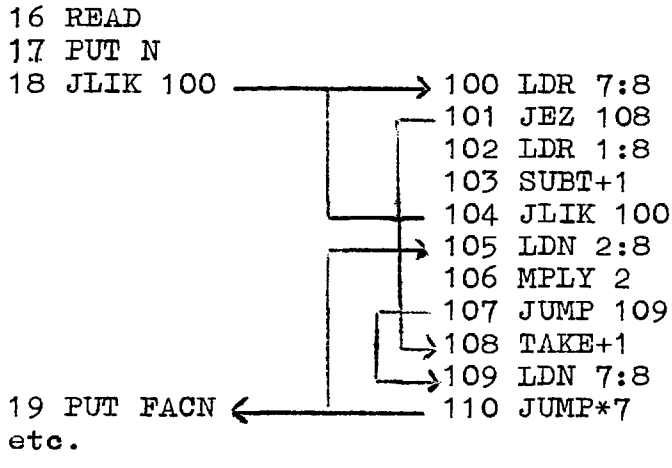
gives the following output when the program is run with 5 supplied as data:

```
*L= 100    +5
*L= 100    +4
*L= 100    +3
*L= 100    +2
*L= 100    +1
*L= 100    +0
*L= 113    +1
*L= 113    +2
*L= 113    +6
*L= 113   +24
*L= 113  +120   FACTORIAL(5)=  +120
```

A stack simplifies the work somewhat. Probably the simplest workable concept is a special cell (with address 8, say) which is the top of a stack. This top is accessed, changed as usual and behaves normally except when it is

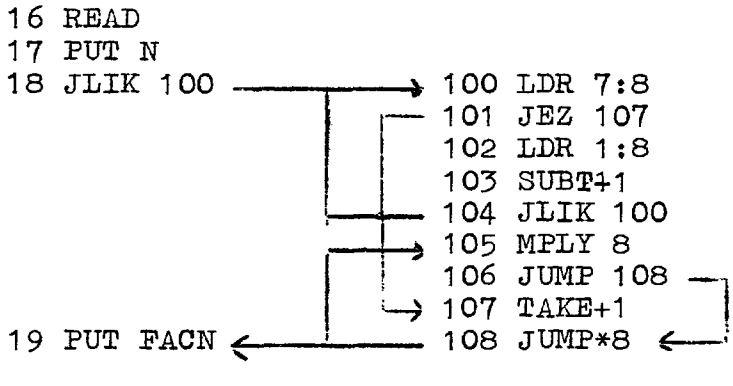
specified as an index register in conjunction with LDR or LDN. As before, LDR has the special property (when used with index 8) of pushing down the current C(8) before assigning a new value, but LDN removes the current top value and leaves the next value in its place. LDN used with an empty stack leaves C(8) = 0.

The factorial example is now repeated using this extension:



The saving is only three instructions.

A less neat, but probably more useful solution is the one proposed which allows a range of instructions to remove items from the stack. This yields the following:



The stack is also used as temporary work space during automatic programming in place of the non-standard method employed in BBC3.

Modification A modification to the library routine READ is to be implemented. This routine is now to search the input buffer area for either a numeric character of a letter. Spaces, new lines and commas are still to be ignored initially and some other characters continue to constitute errors. If the first character the routine inputs is a letter, subsequent letters and digits are input, and the first three and last are left in the accumulator as an S-word. Because buffering enables LIBR routines to 'look ahead', the routine is able to exit with the terminating character as the next to be input. If the first character input is not a letter, the routine behaves as at present and the current alternative of allowing an S-word in quotes is maintained.

Reason Some teachers have felt the need to examine the BBC assembler and have found the assembler written in itself useful but slow. The writing of the City and Guilds compiler has also high-lighted the same limitation.

Modification A special cell (in block 0) is designated as an addressable clock and this is incremented, if negative, immediately before the SCR is incremented during the instruction execution cycle. An interrupt to a new block specified by NBN (next block number) is caused when the clock register is increased to zero.

Reason General Questions, 4, in the Manual makes an attempt at sharing time between two programs. This type of question which is fundamental to the concept of time-sharing, only becomes realistic if some sort of clock is available.

Modification The following extension to the assembler is to be implemented. This new assembler allows labels for forward referencing. Indirect conditional, or indirect unconditional jumps use identifiers in place of absolute or relative addresses thus:

```
JUMP*IDEN
```

Such identifiers are stored at the top end of store in the usual way (but are declared by default without a warning. When an identifier is used to label an instruction it precedes the instructions location thus:

IDEN:85 ADD NEXT

An instruction so labelled causes the compilation of a P-word into the work space associated with IDEN, and if IDEN has not yet been declared it is declared now. In the case illustrated above, the instruction NTHG 85 is stored there so that JUMP*IDEN becomes equivalent to JUMP 85.

Reason Users familiar with traditional assemblers expect this type of facility. It is easy to implement and illustrates a meaningful use of indirect addressing used with jump instructions. The decision to use P-words rather than I-words within the workspace of the identifier enables programs using forward references to be correctly listed and to be relocateable when ADVANCE is used.

Modification The following minor modifications are made in the light of experience using BBC3:

a) JEZ, JNZ, SKAE, SKAN can be used with operands of any type. Their meaning remains unchanged when I- and F-word operands are used, but JEZ and JNZ are now also meaningful with zero or non-zero.P- or S-words, while SKAE and SKAN are meaningful between operands of any type. SKAE and SKAN, used in this context, take 'equal' to mean 'equal including type'. Skip and jump instructions involving 'greater than' and 'less than' remain illegal except when they are used with arithmetic operands.

b) When an error occurs, ERROR only is output, but the full error is available if the user merely types carriage return line feed. Full run time errors now include the display of the content of the offending accumulator, store location (if appropriate) and SCR.

Modification The command lnnguage is modified to fall more in line with Dartmouth Basic and to cope with the extended machine. The most fundamental change is the dropping of apostrophes round commands[2] In the past, when input was required; the computer was either expecting a command, an instruction or data to a running program. Now the first two are combined and the computer looks for a location or a label and expects a command only if it finds neither.

The following commands have also been modified:

a) RUN. Three variants now exist. The first two are as before and the current block is assumed, but the third variant RUN m, n is interpreted to mean run from location m in block n.

b) FETCH and FILE are replaced by LOAD, SAVE and SCRATCH. LOAD must always be used if a program is to use other than blocks 0 and 1 (which are initiated by LOGIN). There may be any number of LOAD commands but blocks must always be loaded before they are used. The following variants are available:

i) LOAD FROM 3 block 1 is copied from file 3

ii) LOAD 2 FROM 3 block 2 is copied from file 3

iii) LOAD 2-4 FROM 3-5 blocks 2 to 4 are loaded from files 3 to 5

iv) LOAD 2 FROM OXO* block 2 is loaded from a systems file (read only) called OXO*. All systems file names end with an asterisk

v) LOAD 2 FROM MINE block 2 is loaded from a user file called MINE

vi) LOAD 2-4 FROM 0 user files are numbered from 1 to 7. Zero specifies that files 2 to 4 are to be loaded with all zeros.

[2] An initial apostrophe is still needed if a command is read within data.

SAVE has a similar set of variants:

i) SAVE IN 3 block 1 is copied to file 3

ii) SAVE 2 IN 3 block 2 is copied to file 3

iii) SAVE 2-4 IN 3-5 blocks 2 to 4 are copied to files 3-5

iv) SAVE HIS IN MINE the block named HIS is first renamed MINE and then replaces the file already called MINE

SCRATCH is used to UNLOAD blocks no longer needed All blocks must be scratched before they are reloaded. The variants, SCRATCH HIS, SCRATCH 3, SCRATCH 3-5 and SCRATCH OXO* are all available, but only core images of blocks are scratched, the only way to remove a file is to save something on top of it.

c) All the other commands are concerned with a single block and this is always block 1 unless another block is specified. The specification of another block takes place immediately before another command is used, and is temporary, since block 1 is the assumed block for any subsequent commands. The command NEWBLOCK followed by a block name or number is used for this purpose.

d) The command NAME is used to name or rename a block. Again block 1 is assumed unless the command is preceded by NEWBLOCK.

e) The command LOGOUT saves block 0, scratches all other blocks and a user must LOGIN again to continue.

10 A COMPARATIVE SURVEY OF SIMILAR MACHINES (REAL AND HYPOTHETICAL)

This chapter attempts to put BBC3 into perspective within the full range of computers (real and hypothetical) which might be used in an introductory course. The concept of special-purpose software and hardware for educational use is not new but it is only very recently that attention has been directed specifically towards the school student. The following would appear to have fairly similar aims to those of BBC3 and will be considered in varying detail according to their apparent merit:

a) City and Guilds Mnemonic Code
b) TAM (used by S.C.A.P.E.)
c) IMDAC (used at the London Institute of Computer Science with M.Sc students).
d) CESIL (used in the ICL CES package)
e) Simulac (designed by a working party of the United Kingdom Co-ordinating Committee for Examinations in Computer Science).
f) IBM Schools' computer
g) Argus 600 (a small real computer)

In each case the design of the hardware (or simulated hardware) and software is assessed in terms of its ability to demonstrate to the student how computers work. This thesis is not concerned with defining the precise concepts that ought to be introduced and in any case such concepts will vary with the ability, interests and academic maturity of the student. At the same time, it might help if the following broad objectives are kept in mind. This list is not exhaustive and it is not intended that any order of merit should be assumed, but it is felt that the computer and associated software should

a) be able to demonstrate how modern computers work at a near hardware level;

b) provide a painless introduction to numeric and non-numeric low-level programming;

c) provide insight into the concept of program translation at an assembly level and at a higher level;

d) be able to demonstrate in a fairly general way some of the modern uses to which a computer might be put (e.g. scientific and commercial programming, timesharing, computer aided learning, simulation, information retrieval).

City and Guilds Mnemonic Code. This was first devised in 1964 for use in the C and G 319 examinations. It was revised in 1968 but these modifications represented no change in philosophy and were mainly concerned with extra refinements.

Basically, as its name implies, C and G Mnemonic Code describes a language rather than a hypothetical computer. Attempts are made to define the machine but this aspect has never been properly thought out and the more discerning student is left with a number of unanswered questions. In particular the detail on the method of storage used for 'numbers' and 'instructions' is missing and students are unable to experiment by assembling instructions via their numeric format or vice versa. Any answers to these questions will be dependent on the particular implementation.

As a tool for teaching elementary mathematical programming the code has been highly successful. The student is provided with a floating point machine which appears to work directly in denary. String handling facilities are provided very much as an afterthought and although backing store is provided, it is difficult to see how any realistic non-numeric or commercial work could be undertaken.

Array work is made possible via a traditional index modification approach. Any of the first 10 of the 1000 words of store can be used as an index but as only floating point arithmetic is available to manipulate the content of these index registers the concept becomes rather confusing.

Because it has not been over ambitious, the implementation of the code on a large range of machines has been possible and the code is consequently widely used.

TAM was first used as long ago as 1963 and represents one of the earliest hypothetical computers used to teach basic computer architecture. It is popular in Scotland where it is used officially within S.C.A.P.E. Its assembler is entirely numeric and is oriented towards card input and line-printer output. By today's standards it is entirely untypical of any real machine and programs written for the assembler are difficult to read. The only aspects which are in its favour are its simplicity and the fact that the machine itself is reasonably well defined.

IMDAC In the introduction to the programming manual A.J.T. Colin states his reasons for the design of the machine as follows: "Like many teaching systems, it avoids the difficulties and disadvantages of using any actual computer by the introduction of a hypothetical, 'quirkless' machine called IMDAC. On the other hand, the system differs from most others of its kind in that IMDAC is not simplified to the extent that it ceases to resemble a real computer, but is a viable machine, with an order code similar to that used by several existing machines. Its viability is illustrated by the fact that the IMDAC symbolic assembly program is itself written in terms of IMDAC machine instruction."

In fact the IMDAC machine turns out to be fairly straightforward in its design. It is true that it is 'quirkless' and that the decision to write the assembler within the hypothetical machine has forced a definition which is clear and unambiguous. The bootstrap process used to load the assembler is convincing to the specialist but it is probable that the beginner will find the verbal description more understandable than reading the details of the programming.

IMDAC provides 4 floating point instructions amongst its order code of 32 functions. 24 bit words are divided into two fields for floating point arithmetic but the absence of the type arithmetic of the BBC machine makes mixed arithmetic tedious and students are known to make many trivial errors at this level which are difficult to diagnose.

Index modification is by one or both of the two accumulators. Immediate and direct addressing is possible but no indurect addressing is available. It seems a pity that the user must specify an immediate operand explicitly in some cases but not in all, for although a particular function is likely to be used more with one type than another, this cannot be easy for the beginner to appreciate.

IMDAC uses purely numeric label referencing within its assembler and this appears to introduce unnecessary confusion between label and address referencing.

All functions are assembled via two letter mnemonics and the primitive system of symbolic addressing makes the source code a rather spartan diet for the beginner.

In fairness, it should be stated that IMDAC was first designed for M.Sc students (Computer Science) in 1964. With this class of student it proved to be successful, but since that time software and hardware developments have made it somewhat uninspired when judged by today's standards.

CESIL This language (there has been no attempt at defining a machine) is introduced into the CES[1] package devised by John Hoskyns and Co Ltd. since taken over by ICL. This package is designed as a one year appreciation course for the more senior secondary school student. Students on the course use the package material and are taught by their own staff, who in turn have attended teachers' courses associated with the package. Practical work in CESIL and 903 SIR is provided by a postal service.

[1] Computer Education in Schools

Since CESIL is only used for a few weeks until the student progresses to the assembler of a real machine its limitations are probably not too inconvenient. At least the format is readable, convincing and very much at the right level for this class of student. The fact that the language is known to be used only via a postal service has influenced its design, and students are clearly not allowed to get too close to the machine.

SIMULAC This is the name given to the machine put forward by a working party of the United Kingdom Co-ordinating Committee for Examinations in Computer Science. Its design has been greatly influenced by the presence on the working party of both the author of this thesis and the author of the Take and Put code from which BBC3 originated.

The working party rejected BBC3 as it stands because it was felt to be too ambitious and too large (both in word length and in its entirety) to be capable of being implemented on the smaller machines. Nevertheless, many of the features that have been successful in BBC3 are present in Simulac. The order code is based on an 18 bit word, using a seven bit function field and a ten bit address field (the eleventh bit is left spare). This generous supply of function potential has resulted in many functions being offered in three modes, viz., immediate, direct and indirect. No index registers are provided and floating point arithmetic is available only as an afterthought.

The two pass assemblers is fairly traditional in outlook but includes a set of very clear mnemonics which can either be given in full or in an abbreviated form.

MULTIPLY.
MULTIPLY.C
MULTIPLY.CC

are the three mnemonics for multiplication using an

immediate operand, a direct and an indirect address (the C stands for 'content of' and the CC for 'content of content of'). Although these mnemonics can be given in full as shown, the assembler is designed so that it ignores all letters after the first three, but before the period, so that MULTIPLY can be shortened to MUL or MULT by those students who prefer the abbreviated form.

Although Simulac and its associated assembler represents an advance on many of its predecessors it suffers from being designed by committee. This has had the effect of eliminating the more exciting features proposed, so that the result tends to be a common denominator of the sectional interests represented. Simulac has not been generally tried out with students and it remains to be seen whether it will be accepted by the main committee.

IBM Schools' Computer It is sometimes argued that the presence of a small computer in a school as opposed to a terminal to a larger machine is likely to have more impact on schoolchildren. Undoubtedly, children find any form of hardware appealing, but past experience seems to demonstrate that once the novelty of the hardware has passed, it is its intrinsic worth which determines the success of any teaching aid.

The IBM Schools' Computer was conceived as a piece of hardware within a school which would use a standard TV set as a display device and a standard tape recorder as backing store. With its touch keyboard it has a number of hardware features which make sense within the school environment. However, within this thesis it is only appropriate that it should be discussed from the same standpoint as the alternatives. After all, it is hoped that the man-machine interface will be subjected to close scrutiny generally, and that education will be able to reap any benefits from an overall improvement in techniques.

The computer (from the student's point of view) contains two segments, each of a hundred eight-denary digit words (strictly speaking storage is in BCD). A novel but entirely convincing form of denary floating point arithmetic, which reduces to integer form where possible, is used in conjunction with a three address code. The machine has sufficient redundancy to make error detection easy and younger students seem to find the entirely numeric program input digestible.

The machine has no capacity for immediate operands and without an effective assembler, constants within programs are tedious to handle. Indirect addressing is possible, but there is only one form available so that if one address is to be accessed indirectly, so must the other two.

Used in this way, the computer is really seen at its extracode level, but more senior students can descend to a more fundamental one address machine and have found it challenging to rewrite the extracodes.

The chief drawback to the machine would appear to be its non-standard approach. Many of the programming tricks that are learnt are not generally applicable and although some students are happy never to progress beyond the entirely numeric stage, it is perhaps unfortunate that they will finish a course using the computer without first hand experience of alphabetic input or output and without appreciating the need for program translation.

At the time of writing the future of the machine is in doubt. IBM have stated that they never intended to go into full-scale production and have no plans for its manufacture while other prospective manufacturers would find it difficult to produce the machine at a realistic price.

Argus 600 This small Ferranti computer has been suggested as a machine which might be installed into a school to provide a basic programming tool at the level of the other machines considered in this chapter. The manufacturers state that it is designed for a dual purpose. As a controller, it is claimed that it can be used for a variety of machine control applications. As a computer, it is envisaged as a stand alone research machine or as a satellite in a larger Argus 400 or 500 system. It is in its capacity as a stand-alone computer that it is considered here.

A number of options are available. Those chosen represent what a school would be lucky to afford but what would enable it to perform in a similar way to its competitors within this thesis. The proposed configuration is given below:

MAC64	Processor with monitor panel	£1685
MAC62	2 times 1K store block	£ 750
MAC63	Store expander with additional power unit	£ 350
MAC66	Teletype drive card	£ 250
		£3035
	Teletype model 33	£ 515
	Total	£3550

The assembler ASSIST is oriented towards a 2K machine and although mnemonics are provided for the 17 different functions the assembler remains very primitive. All addresses must be given in octal and no label referencing is provided.

The instruction word uses a 3 bit function field and a 5 bit address field with the address referring either to the current page or to page zero. A double length jump instruction allows a thirteen bit address for transfer between pages and a group of addressless functions

include a pair of conditional skip instructions. No hardware address modification or indirect addressing is provided so that software modification must be used and this would quickly become tedious.

On the credit side, the whole machine could be understood by quite young children and the monitor panel provides excellent opportunities for students to gain insight into what is happening within the machine.

The BBC alternative Judged against the previous summaries both BBC3 and BBC-10 appear very comprehensive. The order in which the various topics need to be introduced has been carefully chosen however, and it is possible to introduce a sophisticated machine to quite immature students. Probably the most important feature which has contributed to the success of BBC3 at this elementary level over any other machine is the manner in which the type arithmetic has been exploited to allow floating and fixed point arithmetic to be mixed and to allow sufficient redundancy for really good run time diagnostics.

A machine comparison

Attributes	Machine and associated software								
	BBC -10	BBC 3	C&G 319	TAM	IMDAC	CESIL	SIM-ULAC	IBM Soh	ARGUS 600
Directly addressable store size	1024	1024	1000	1000	8192	?	1024	200	32
Extendable?	yes	no	no	no	no	?	no	no	yes
Binary/Denary	B	B	D	D	B	?	B	D	B
Word length	24[1]	24[1]	7	5	24	?	18	8	8
Integer arithmetic possible?	yes	yes	no	yes	yes	yes	yes	yes	yes
Floating point arith. possible?	yes	yes	yes	no	yes	no	yes	yes	yes
Character handling capabilities?	yes	yes	yes[2]	no	yes	no	yes	no	yes
Immediate operands for most functions	no	no	yes	no	yes	no	yes	no	no
Indirect addressing	yes	yes	no	no	no	no	yes	yes	no
No. of index reg.	16	16	10	1	2	0	0	0	0
Software modification possible?	yes	yes	no	yes	yes	no	yes	yes	yes
No.of accumulators	4	2	1	1	2	1	1	0	1
Special purpose reg. addressable	yes	yes	yes	no	no	no	yes	-	no
Error interrupts under prog.control	yes	no	yes[2]	no	no	no	no	no	yes
Well defined machine	yes	yes	no	yes	yes	no	yes	yes	yes
Function mnemonics	good	good	poor	none	bad	good	v.g.	none	poor
Labels available for forward ref.	yes	no	no	no	yes	yes	yes	no	no
Full symbolic addressing	yes	yes	no	no	no	yes	no	no	no
1 or 2 pass assem.	1	1	1	1	2	2	2	-	1
Literals possible	yes	yes	no	no	no	yes	no	no	no
Trace facilities	yes	yes	yes[2]	no	yes	no	yes[2]	yes	yes
Post-mortem possible	yes	yes	no	yes[2]	no	no	yes	yes	yes
High level facilities	yes	yes	no	no	no	no	no	no	no
Bootstrap input	no	no	no	no	yes	no	no	-	yes
Usual/alternative assb. within mach.	alt	alt.	none	none	usu.	none	alt.	-	usu.

1. Plus two type bits. 2. Very limited.

11 CONCLUSIONS

In 1966 when BBC1 was first introduced at Hatfield School, computer education in schools was still a new idea. Now, some four years later, it is still by no means commonplace, and Hertfordshire is the only local authority to take seriously the concept of on-line access for schoolchildren. The initial PDP-10 configuration at Hatfield is shown in appendix 2 fig. 6. The six schools named here are to have unlimited access to the computer and this is only the first phase of a development system. There are a number of other schools who are seeing the importance of computer education within the curriculum and who will reply on a courier service to start with.

It seems reasonable to assume that a terminal within a school will be used in three broad overlapping areas. As a tool within the mathematics syllabus the computer terminal has already proved itself in American high schools and in this country the compter work in the School Maths Project (SMP) and Mathematics in Education and Industry (MEI) is being accepted.

A second but so far less well established area of development is in computer aided learning. Here, the emphasis in the future is likely to be in the more fruitful areas of simulation, games and computer managed instruction, rather than in the more traditional individualised CAI approach. Viewed in this way, the computer is seen more as a part of the total development of educational technology enriching teaching techniques, rather than supplanting existing methods of education.

The third area is that of computer science. The Schools' Committee of the B.C.S. in their document 'Computers for All' emphasise the importance of computer education for all secondary school pupils. These pupils need to learn enough about the computer itself and about its applications to enable them to grow into useful citizens in an automated

age. Clearly a number of young people taking such courses are going to seek options within the school curriculum that will provide them with a more detailed knowledge of computer science. Perhaps the newly emerging computer science 'A' level examinations will provide the basis of such courses but in any case Computer Science degrees are now well established and students who are considering a profession within the computer industry must be allowed to discover what is involved in such a step.

There is as yet no established pattern for computer courses at school level but it is the author's belief that students should be brought into contact with the computer by the age of 13 or 14. At this age it is contended that the correct level of approach is that supplied by BBC3 where the concepts of a computer store, a stored program and program translation can be clearly demonstrated. For the non-specialist who progresses to general algorithm formulation, some higher level language will probably be introduced at an early stage, but even this student is going to gain from an understanding of elementary compilation techniques.

The more advanced computer science student will increasingly be faced in the future with the problem of having to understand the intricacies of third generation computers from a machine code standpoint. It is argued that such students will find enlightening the early study of the same concepts in the carefully controlled environment provided by the special purpose BBC machine.

It has been shown that at a number of different levels, BBC3 is able to provide the school student with a worthwhile learning aid. If BBC-10 is able to draw on its 2000 student terminal hours of experience, it should be possible to achieve for the PDP-10 at Hatfield a system which can cover an even greater span of the educational market with at least equivalent success.

ERROR SUMMARY. USER 1,2,3

NAME	LOGIN		INSTR	EOP	ANS	ERROR
VERA	2	0	30	2	2	3
YVOE	2	0	64	3	3	0
AVEY	3	0	31	3	3	4
RIDR	2	0	21	3	3	2
SMIH	4	0	29	2	6	16
PITR	3	0	24	4	4	2
EDNA	3	0	49	4	4	5
CHAE	3	0	46	4	4	3
THOS	3	0	39	5	5	7
GRAM	2	0	31	2	2	0
MARY	2	0	31	3	3	1
COLR	3	0	38	2	2	2
VALE	2	0	27	3	3	0
DARY	2	0	32	2	2	4
PATA	2	0	34	2	2	0
ALAN	0	0	0	0	0	0
BILL	3	0	0	0	0	4
MEL	0	0	0	0	0	0

~

An example of the Error Summary output.

An Example of an On-line Session.

The user has typed the lines that start with a question mark or an apostrophe.

```
'LOGIN' BILL

BILL
?'DECLARE' A,B,C,ANS,N
?16 READ
?17 PUT A
?18 READ
?19 PUT B
?20 READ
?21 PUT C
?'LOOPV' N:=1:1:6

  22 JOI 23
  23 TAKE 1013      (    +1   )
  24 PUT 1014       (N)
  25 TAKE 1013      (    +1   )
  26 PUT 1010       (BBC1)
  27 TAKE 1007      (    +6   )
  28 LIBR 12        (INT)
  29 PUT 1008       (BBC2)
  30 JUMP 33
  31 TAKE 1010      (BBC1)
  32 XADD 1014      (N)
  33 DECR 1008      (BBC2)
  34 JEZ 0
?'ASSIGN' S:=A+&&(A+B+C)/2.0

ERROR NO.12 IDENT. NOT DECLARED L=  34
?'DECLARE' S
?'ASSIGN' S:=A&(A+B+C)/2.0

  35 JOI 36
  36 TAKE 1022      (A)
  37 ADD 1020       (B)
  38 ADD 1018       (C)
  39 DVD 1003       (+2.0000   )
  40 PUT 1004       (S)
?'ASSIGN' ASN&&NS:=PRINT(SQRT(A&S*(S-A)*(S-B)*(S-C)))

  41 JOI 42
  42 TAKE 1004      (S)
  43 SUBT 1022      (A)
```

Code generated by 'LOOPV' command. The exit address at 34 is inserted later.

generated by 'ASSIGN' command

```
  44 PUT 1000        (BBC3)
  45 TAKE 1004       (S)
  46 MPLY 1000       (BBC3)
  47 PUT 1000        (BBC3)
  48 TAKE 1004       (S)
  49 SUBT 1020       (B)
  50 PUT 998         (BBC4)
  51 TAKE 1000       (BBC3)
  52 MPLY 998        (BBC4)
  53 PUT 1000        (BBC3)
  54 TAKE 1004       (S)
  55 SUBT 1018       (C)
  56 PUT 998         (BBC4)
  57 TAKE 1000       (BBC3)
  58 MPLY 998        (BBC4)
  59 LIBR 1          (SQRT)
  60 LIBR 5          (PRIT)
  61 PUT 1016        (ANS)
?62 INCR C
?63 LINE
?'REPEAT' N

  64 JUMP 31
?65 STOP
?'RUN'

DATA?3 4 2
+2.9048
+4.4722
+5.5623
+6.0000
+5.3328
+.00000

END OF PROGRAM
'FILE' 2
TRANSFER COMPLETE
```

Generated by the second 'ASSIGN' Command.

Generated by 'REPEAT' to close the loop. The instruction in 34 is now amended to JEZ 65

Results

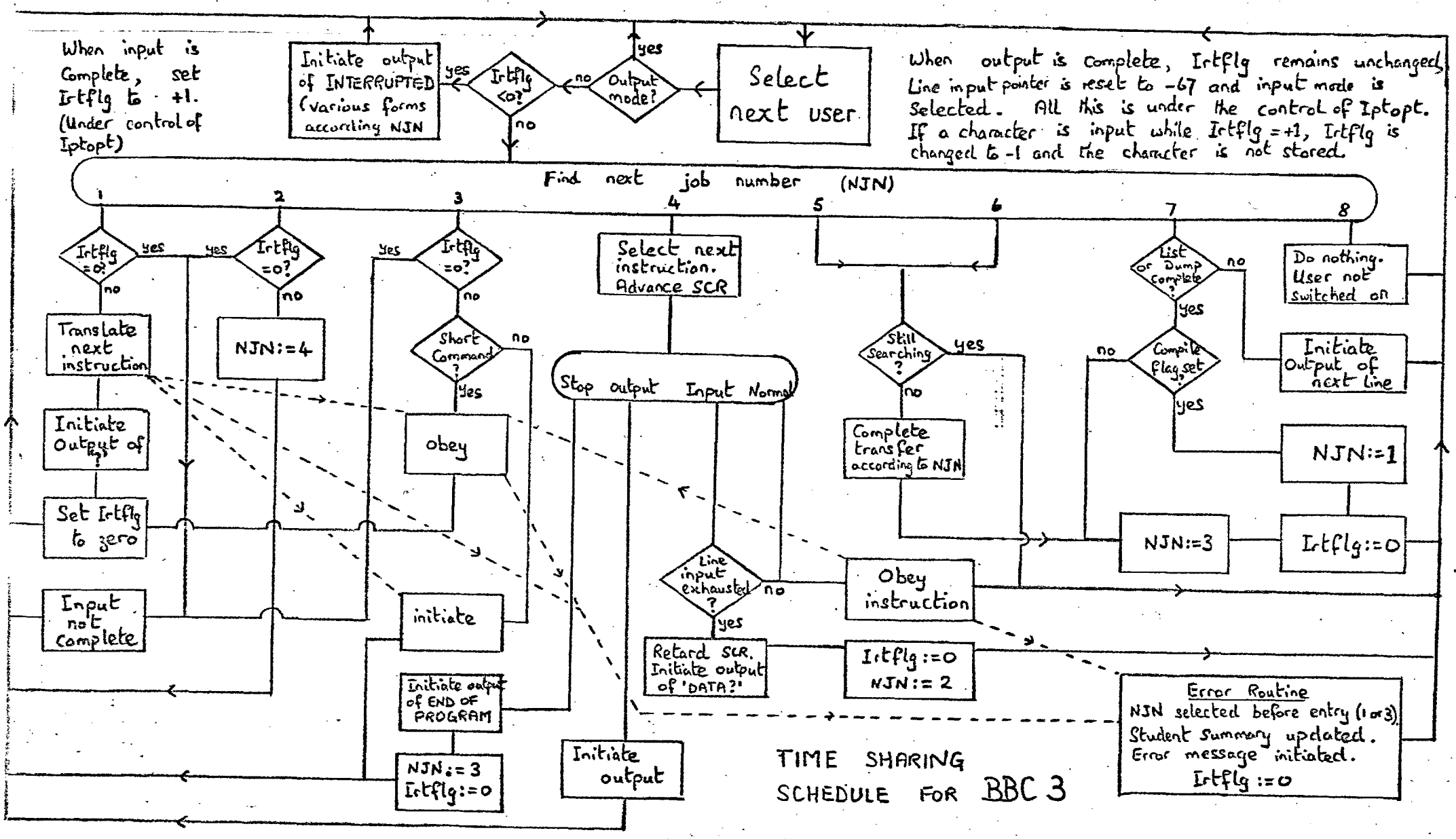
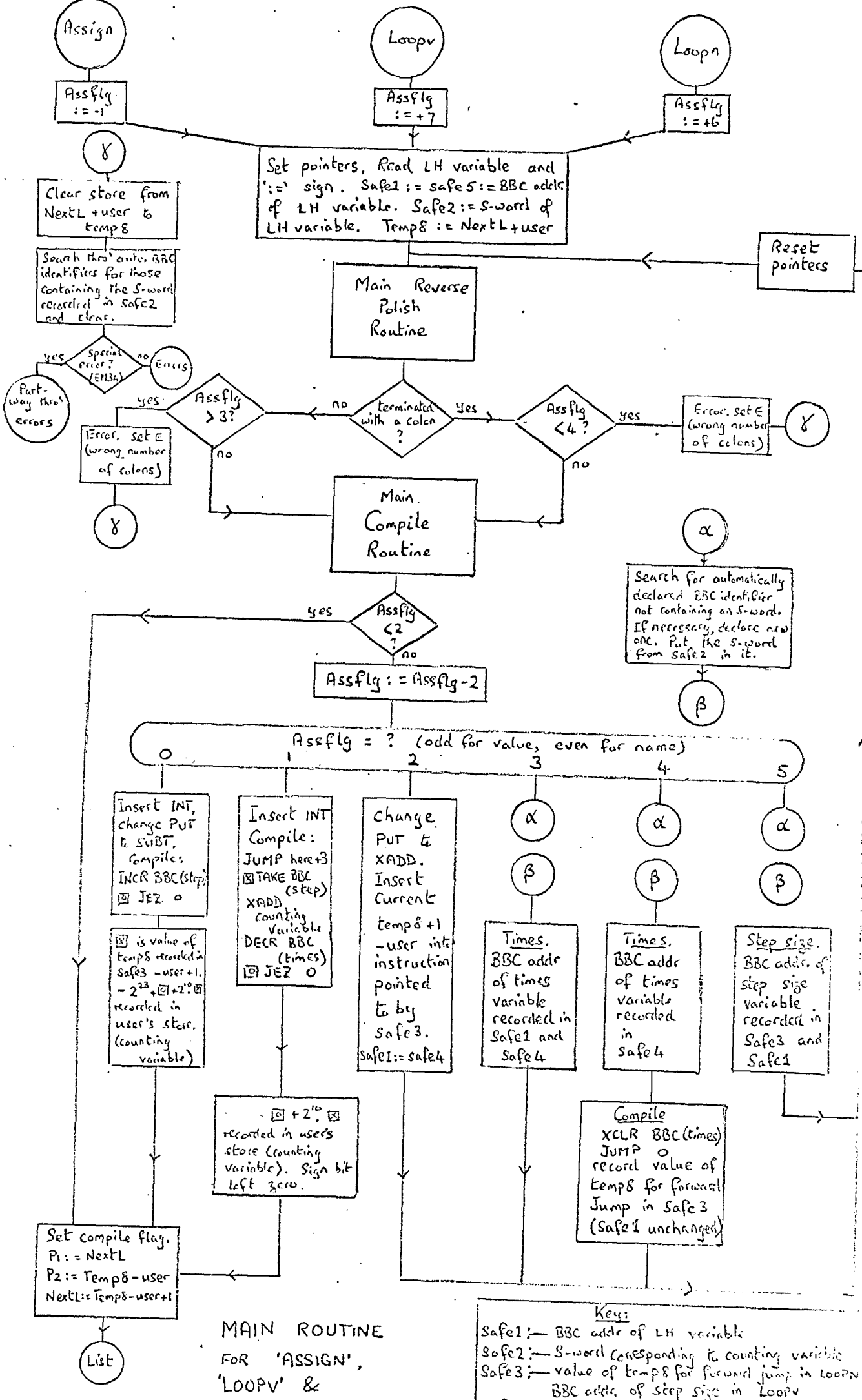

When input is Complete, set Irtflg to +1. (Under control of Iptopt)
Initiate output of INTERRUPTED (various forms according NJN
Irtflg <0?
Output mode?
Select next user
When output is complete, Irtflg remains unchanged. Line input pointer is reset to -67 and input mode is selected. All this is under the control of Iptopt. If a character is input while Irtflg = +1, Irtflg is changed to -1 and the character is not stored.
Find next job number (NJN)
1
2
3
4
5
6
7
8
Irtflg =0?
Translate next instruction
Initiate Output of ?
Set Irtflg to zero
Input not complete
Irtflg =0?
NJN:=4
Irtflg =0?
Short Command ?
obey
initiate
Initiate output of END OF PROGRAM
NJN:=3 Irtflg:=0
Select next instruction. Advance SCR
Stop output Input Normal
Line input exhausted ?
Retard SCR. Initiate output of 'DATA?'
Initiate output
Still Searching ?
Complete transfer according to NJN
Irtflg:=0 NJN:=2
Obey instruction
List or Dump Complete ?
Compile flag set ?
NJN:=3
NJN:=1
Irtflg:=0
Do nothing. User not switched on
Initiate Output of next line
Error Routine
NJN selected before entry (1 or 3).
Student Summary updated.
Error message initiated.
Irtflg:=0
TIME SHARING SCHEDULE FOR BBC 3

Fig. 1



MAIN ROUTINE FOR 'ASSIGN', 'LOOPV' & 'LOOPN'.

Fig 2

'LOOPN' ANS := D*D+4*C : 2*B-SQRT(D) : C/(C-B)

} The user himself now enters the statement(s) under the control of the Loop using 'ASSIGN', further loop instructions or hand coding.

'REPEAT' ANS

Assflg=6

Addr of Ans → safe1 & safe5. <ANS> → safe 2

```
51 TAKE D
52 MPLY D
53 PUT BBC1
54 TAKE+4
55 MPLY C
56 PUT BBC2
57 TAKE BBC1
58 ADD BBC2
59 PUT ANS
```

Initial Value

Ans := D*D + 4*C using Main Routine

Assflg=4

```
60 XCLR BBC1   } Fixed Code
61 JUMP 68
```

Search for times (BBC1)

Addr of BBC1 → safe4
safe3 := 61 (+user etc)

```
62⊠TAKE+2
63 MPLY B
64 TAKE2 D
65 SQRT2
66 SUBT 2
67 PUT ANS
   Xadd
```

Step Size

Ans := 2*B − SQRT(D) using Main Routine

change Put to Xadd

Assflg=2

Insert 68 (use safe3) Addr of BBC1 from Safe4 to Safe1

```
68 TAKE C
69 SUBT B
70 PUT BBC2
71 TAKE C
72 DVD BBC2
   73 Int
74 Subt
75 PUT BBC1
```

Times

BBC1 := C/(C−B) using Main Routine

Insert Int and change Put to Subt

Assflg=0

Ans := -2^{23} + ⊠.2^{10} + ⊡

```
75 INCR BBC1 (times. Addr from Safe 4)   } Fixed code (with variable addresses)
76⊡JEZ ⊠93
```

'REPEAT' ANS

```
92 JUMP 62
```

⊠ and ⊡ unpacked out of ANS and used here and here

Fig 3

```
'LOOPV' ANS := D*D+4*C : 2*B-SQRT(D) : C/(C-B)
              }  The user himself now enters the statement(s) under the control
              }  of the loop using 'ASSIGN', further loop instructions or hand coding.
'REPEAT' ANS
```

Assflg = 7

Addr of Ans → safe1 & safe5. <ANS> → safe2

```
51 TAKE D
52 MPLY D
53 PUT BBC1
54 TAKE+4
55 MPLY C
56 PUT BBC2
57 TAKE BBC1
58 ADD BBC2
59 PUT ANS
```

Initial Value

Ans := D*D + 4*C using Main Routine

Search for step size (BBC1) Assflg = 5

Addr of BBC1 → safe1 & safe3

```
60 TAKE+2
61 MPLY B
62 TAKE2 D
63 SQRT2
64 SUBT 2
65 PUT BBC1
```

Step Size

BBC1 := 2*B − SQRT(D) using Main Routine

Search for times (BBC2) Assflg = 3

Addr of BBC2 → safe1 & safe4

```
66 TAKE C
67 SUBT B
68 PUT BBC3
69 TAKE C
70 DVD BBC3
   71 Int
72 71 PUT BBC2
```

Times

BBC2 := C/(C−B) using Main Routine

Insert Int Assflg = 1

Ans := ⊠ . 2^{10} + ⊡

```
   73 JUMP 76    (ie 3 forward)
→  74⊡ TAKE BBC1  (step size. Addr from safe3)
   75 XADD ANS   (counting variable. Addr from safe5)
→  76 DECR BBC2  (times. Addr from safe4)
   77⊡ JEZ Q94
```

Fixed code (with variable addresses)

```
   93 JUMP 74
```

'REPEAT' ANS

⊠ and ⊡ unpacked out of ANS and used here and here

Fig 4

Elliott floating point hardware used with BBC operands

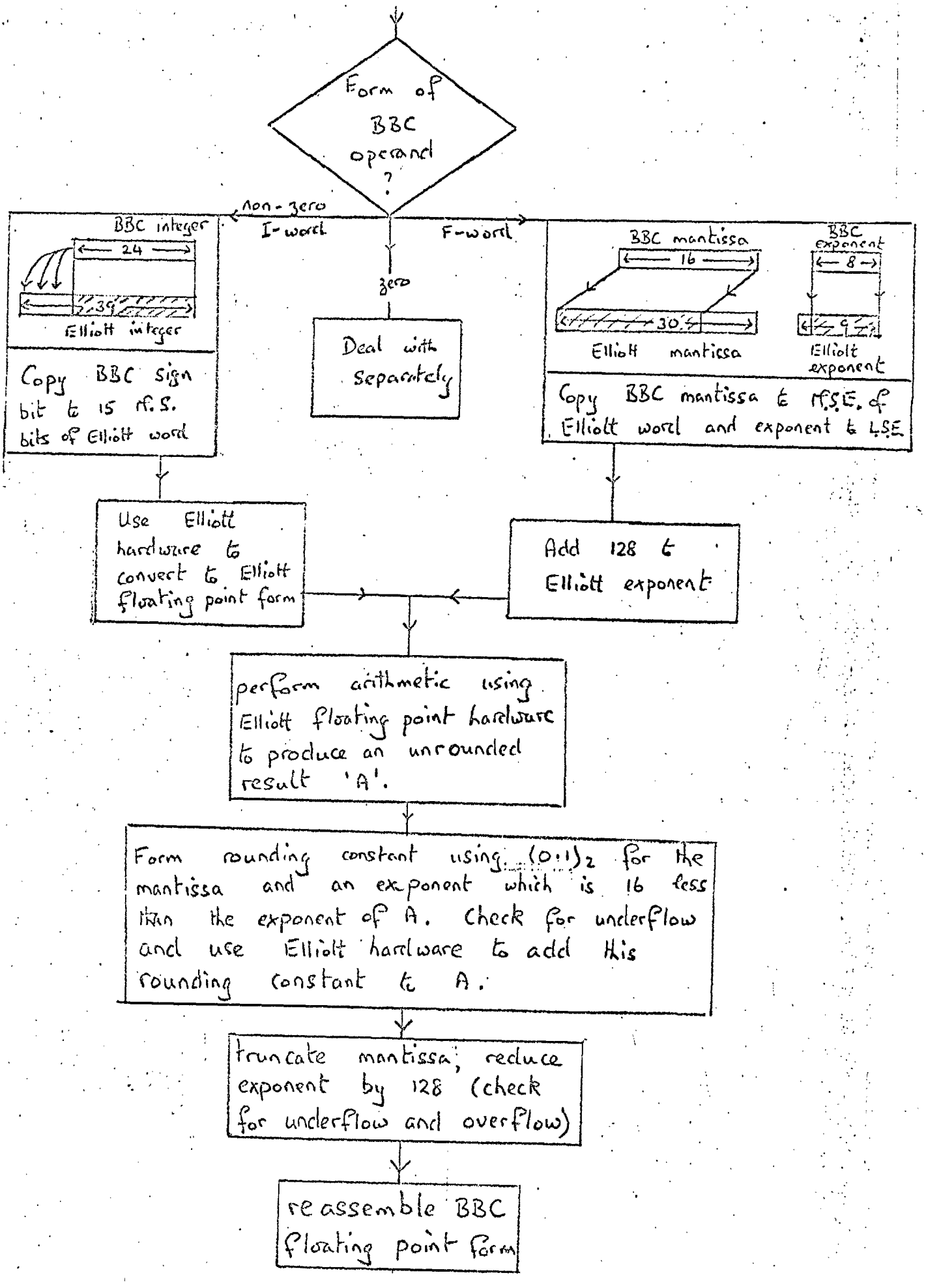
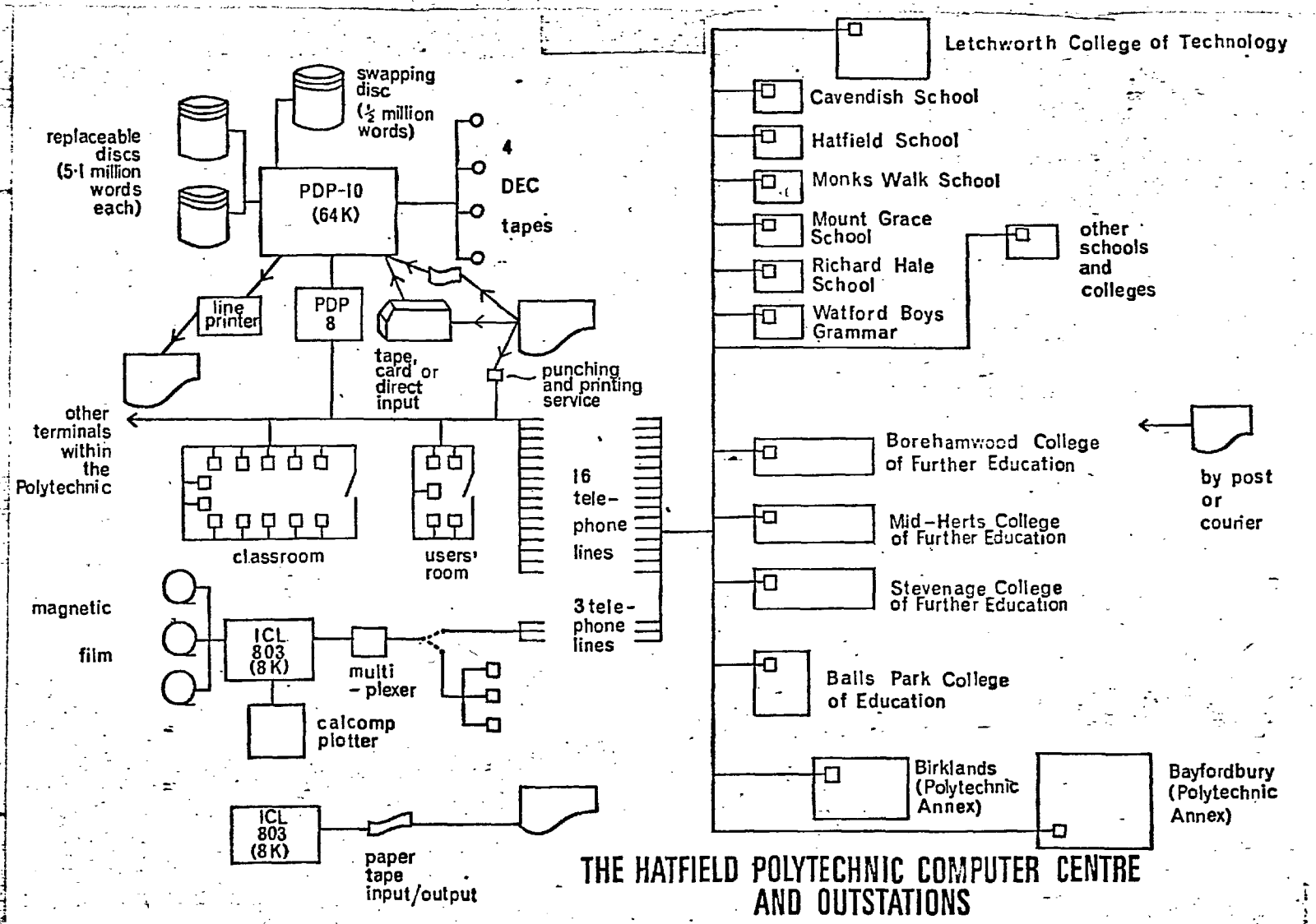


Fig 5



THE HATFIELD POLYTECHNIC COMPUTER CENTRE AND OUTSTATIONS

Fig. 6.

THE HATFIELD POLYTECHNIC

COMPUTER CENTRE

THE 803B MULTIPLEXER SYSTEM

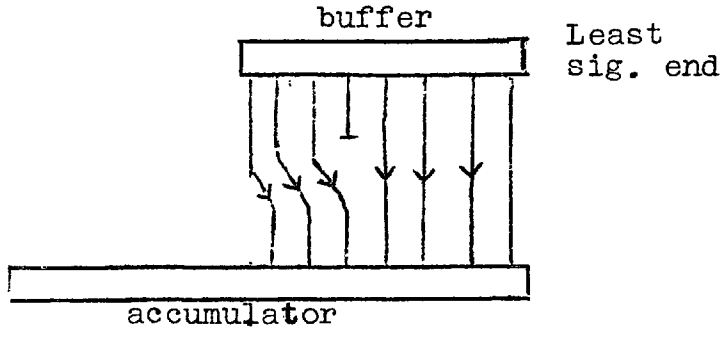
A multiplexer has been manufactured and interfaced to one of the Elliott 803s (computer No.1) by Direct Data of Welwyn Garden City. It is fully tested and is operational. It is controlled by an extended order code using the 75 instruction. A complete summary of the machine code instructions and their functions is set out in Appendix 1. For users who have no knowledge of machine code, a set of Algol procedures have been written and are available via a precompiled package for 5 hole Algol (issue5); details are shown in Appendix II.

The multiplexer can be used to service up to 3 teletype 33 outstations simultaneously. These can be situated locally within the computer centre, remotely - one via a land line and two via G.P.O. telephone lines - to other educational establishments, or any combination of these. Users of the system must make sure that they know how to operate the hardware.

Philosophy Basically the device is operated by a program in core which scans a set of single character buffers at regular intervals (at least 10 times/sec. is recommended).

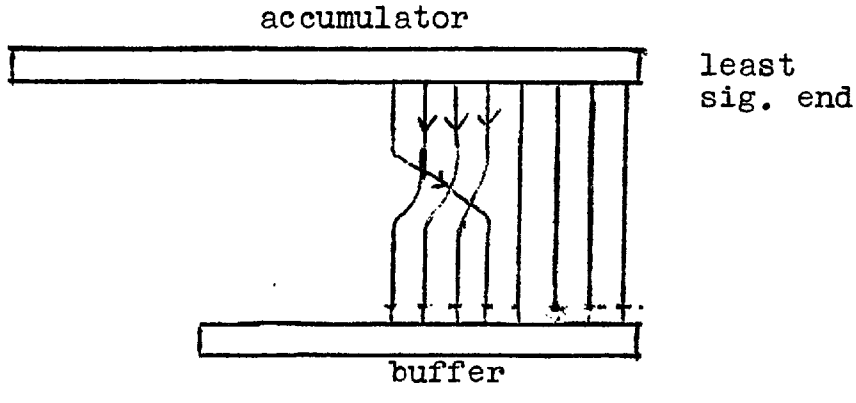
Outstations (or 'users') can be set individually to input or output mode. When in input mode, characters input from the teletype are only typed back when they have been removed from the buffer, this way their successful transmission is confirmed. On input, the fifth channel is regarded as a parity bit and is removed (without checking). On output, this parity bit is not generated automatically; it must be supplied to the buffer in channel 8. A table of character input and output values is given in Appendix III.

Input



Input is into the bottom end of a cleared accumulator.

Output



Transfer does not clear the accumulator.

APPENDIX I

Status A two-bit status word is associated with each user. The values of this word depend on the condition of the buffer and on the current user mode. The table shows the values (in binary) of this word for the various states.

	buffer full	buffer empty
Input mode	01	00
Output mode	10	11

The least significant bit indicates whether the buffer is in a suitable state for transferring data and the second least significant bit indicates the current mode.

A one-bit status word is associated with the master unit. This bit is a zero if no user is ready for data transfer and is a one otherwise

Order Code (time : 576 μs)

75	4097	Transfer to or from buffer	user 1
75	4098	ditto	user 2
75	4099	ditto	user 3
75	6144	Read status of master unit	
75	6145	ditto	user 1
75	6146	ditto	user 2
75	6147	ditto	user 3
75	7169	Set to input mode	user 1
75	7170	ditto	user 2
75	7171	ditto	user 3
75	7681	Set to output mode	user 1
75	7682	ditto	user 2
75	7683	ditto	user 3

APPENDIX II

The following Algol procedures are available for use with 5 hole or 8 hole Algol.

```
multiplexer algol package;
begin integer one,two;
Boolean procedure test(x);  value x;  integer x;
  comment x=0 specifies a test of the multiplexor status and test
  takes the value true if at least one user requires servicing and
  false otherwise. for x=n (n from 1 to 3),test takes the value
  true if user n is ready for a transfer and false otherwise. if
  x takes any other value, test is not defined;
  begin
     elliott(0,0,x,1,7,5,6144);
     elliott(0,3,one,0,2,0,x);
     test:= x ≠ 0
  end of test;

Boolean procedure inmode(x);  value x;  integer x;
  comment x specifies the user, inmode takes the value true if that
  user is in input mode and false otherwise;
  begin
     elliott(0,0,x,1,7,5,6144);
     elliott(0,3,two,0,2,0,x);
     inmode:= x=0
  end of inmode;

procedure chmode(x,y);  value x,y;  integer x,y;
  comment this procedure changes the mode of user x as follows:
  y=0 changes to input mode and y=1 changes to output mode;
  begin
     if  y=0 then elliott(0,0,x,1,7,5,7168)
             else elliott(0,0,x,1,7,5,7680)
  end of chmode;

procedure trans(x,ch);  value x;  integer x,ch;
  comment a character is transferred from ch to the buffer (in
  output mode) or from the buffer to ch (in input mode). the mode
  of the user does not change. x specifies the user;
  begin
     if inmode(x) then begin elliott(0,0,x,1,7,5,4096);
                             elliott(2,0,x,0,0,0,0);
                             ch:=x
                       end
                  else begin integer y;  y:=ch;
                             elliott(3,0,y,0,0,0,0);
                             elliott(0,0,x,1,7,5,4096)
                       end
  end of trans;

Boolean procedure mpxrts;
  comment equivalent to test(0);
     begin integer temp;
           elliott(7,5,6144,0,2,0,temp);
     mpxrts:= temp ≠ 0
     end of mpxrts;
```

```
procedure parity(ch);   integer ch;
  comment takes a 7 bit character from ch and adds in a parity bit
  in channel 8;
  begin integer temp,one;
     temp:= ch;   one:=1;
     elliott(3,0,temp,0,5,1,1);
     elliott(2,4,temp,0,5,1,1);
     elliott(2,4,temp,0,5,1,1);
     elliott(2,4,temp,0,5,1,1);
     elliott(2,4,temp,0,5,1,1);
     elliott(2,4,temp,0,5,1,1);
     elliott(0;4;temp,0,2,3,one);
     if one=1 then ch:=ch+128;
  end of parity;

one:=1;   two:=2;
precompile;


testing multiplexer. package;
begin integer user ,in,out1,out2,char;
     switch ss:=l1,l2;l3,start;loop,transfer,loop2;
start: for user:=1 step 1 until 3do chmode(user,0);
loop:  if not mpxrts then go to loop;
       for user:=1 step 1 until 3 do if test(user) then go to ss[user];
       print fail ;   stop;
l1:    in:=1;   out1:=2;   out2:=3;   go to transfer;
l2:    in:=2;   out2:=1;   out1:=3;   go to transfer;
l3:    in:=3;   out1:=1;   out2:=2;
transfer: chmode(out1,1);   chmode(out2,1);
       trans (in,char);   parity(char);   trans(out1,char);
                                          trans(out2,char);
loop2: if test(out1) and test(out2) then go to start
                                    else go to loop2;
end of test program;
end of multiplexer package;
```

APPENDIX III

T E L E T Y P E 33 P A P E R T A P E C O D E

Tape Code	Dec. Value on input	Dec. Value on output	Char.	Tape Code	Dec. Value on input	Dec. Value on cutput	Char
00001.001	9	9	H.Tab	10001.101	77	77	CR
00001.010	10	10	L.F.	10100.000	80	80	SP
00110.000	16	144	Ø	10110.001	81	209	1
00100.001	17	17	↑	10110.010	82	210	2
00100.010	18	18	↓	10100.011	83	83	#
00110.011	19	147	3	10110.100	84	212	4
00100.100	20	20	→	10100.101	85	85	%
00110.101	21	149	5	10100.110	86	86	&
00110.110	22	150	6	10110.111	87	215	7
00100.111	23	23	'	10111.000	88	216	8
00101.000	24	24	(	10101.001	89	89	)
00111.001	25	153	9	10101.010	90	90	*
00111.010	26	154	:	10111.011	91	219	;
00101.011	27	27	+	10101.100	92	92	,
00111.100	28	156	<	10111.101	93	221	=
00101.101	29	29	-	10111.110	94	222	>
00101.110	30	30	.	10101.111	95	95	/
00111.111	31	159	?	11000.000	96	96	@
01010.000	32	160	P	11010.001	97	225	Q
01000.001	33	33	A	11010.010	98	226	R
01000.010	34	34	B	11000.011	99	99	C
01010.011	35	163	S	11010.100	100	228	T
01000.100	36	36	D	11000.101	101	101	E
01010.101	37	165	U	11000.110	102	102	F
01010.110	38	166	V	11010.111	103	231	W
01000.111	39	39	G	11011.000	104	232	X
01001.000	40	40	H	11001.001	105	105	I
01011.001	41	169	Y	11001.010	106	106	J
01011.010	42	170	Z	11011.011	107	235	⊏
01001.011	43	43	K	11001.100	108	108	L
01011.100	44	172	~	11011.101	109	237	⊐
01001.101	45	45	M	11011.110	110	238	↑
01001.110	46	46	N	11001.111	111	111	O
01011.111	47	175	←				

THE HATFIELD POLYTECHNIC

COMPUTER CENTRE

Revised Operating Instructions for BBC III

The system can now be operated in 2 modes:

1) on typing 'PTAPE' tape is read without operator intervention.
2) On typing 'PTAPE' the user no. + binary 16 is output on the punch as an indication to the operator to input tape.

The 2 methods of operation are described below for the particular cases of Timeshare I and Timeshare II.

TIMESHARE I (no operator intervention for 'PTAPE')

1. Place compiler tape or trigger in reader.
2. Enter 40 0 : 00 0 R.O.N.O.
3. Place tape labelled "Read Command" in reader.
4. Enter 44 5 : 00 0 R.O.N.O.
5. Any other corrections should be entered or
   44 5 : 00 0
6. Place "master-name" tape in reader.

   Enter 40 5 : 00 0
7. Clear keyboard from bottom up.
8. If user 1 and/or user 2 have not supplied tapes to be input during the session, reader 1 and/or reader 2 should be loaded with tape with 'Telep' cr.lf. punched on it 2 or 3 times.
   If tapes are to be input these should be placed in the appropriate reader, i.e. user 1 = reader 1, user 2 = reader 2.
   If user 3 wishes to read tape this will be read from the reader on the side of their Teletype.

TIMESHARE II

The system is initialised in the same way as Timeshare I except that steps 3 and 4 are omitted.

INPUT VIA PAPER TAPE

All tapes should be terminated by the command word 'TELEP' and carriage return, line feed.
To switch to the tape reader proceed as follows:

a) Enter tape into reader with mode button down and load. The tape must always be loaded correctly before step b) is performed because while the keyboard is not clear the other users will be held up.

b) Change to negative word on keyboard (i.e. set up 40 on function 1).

/..

c) Set 1, 2, 3 on address 2 to specify user and press 2048 if reader 2 is required.

d) Clear funtion 1 and then set up 40 again on function 1.

e) Clear address 2.

f) Clear F1.

If continuous output occurs, an attempt has been made to switch two users to the same reader. Go back to step b). Once a reader has been assigned to a Teletype it remains there until it is cancelled by the command workd 'TELEP' on paper tape.

IN CASE OF DIFFICULTY

If any difficulty arises during the session the following procedure should be tried

1) set up 40 6 : 00 0
R.O.N.O.

2) Clear A1 Clear F1.

G. Fletcher

GF/RA
29.5.70

THE HATFIELD POLYTECHNIC

BBC3

(a hypothetical computer)

DEPARTMENT OF
COMPUTER SCIENCE

THE HATFIELD POLYTECHNIC

Department of Computer Science

BBC 3 (a hypothetical computer)

A user's manual written by

W. TAGG

Printed and published by The Hatfield Polytechnic

PREFACE

This book describes a hypothetical computer which has been simulated on one of the Elliott 803s at The Hatfield Polytechnic. As computers are now sophisticated pieces of machinery, they are not suitable as vehicles for introducing Computer Science from a machine code standpoint. Because of this,. it has become fashionable to initiate this aspect of the work via a simple but non-existent computer and to provide practical work via a simulation of this machine on a real computer. One advantage which such a solution offers is that the real computer can provide much more in the way of helpful diagnostics.

In the past, such hypothetical computers have fallen into one of two classes Either they have been based on an ill-defined computer (like the code used for the City and Guilds 319 examination) or they have been so simple that they have not been suitable for problem solving. BBC3 (Beginners' basic computer, mark 3) attempts to bridge the gap between these extremes. This has been achieved by choosing a computer which remains suitable for an initial course but which is, perhaps, more advanced in its design than its contemporaries.

Two programming languages are provided. A fairly standard assembly code is embedded into a command language enabling the user to program in an on-line environment. This command language has been extended to provide high-level features not usually associated with this type of study; the student is provided with a powerful tool but does not loose sight of the machine oriented aspect of the work.

BBC3 was developed from the Take and Put code used in the television series 'Mathematics in Action, Logic and the Computer'. These programmes have been shown each year since 1966 when they were first written and an early version of BBC was used with a National Extension College course associated with the T.V. series.

All of the ideas incorporated into BBC3 have been fully tested in the classroom. During the last three years the system has been used by Computer Science undergraduates, 'A' level Computer Science students, sixth form 'arts' students, 11 year old school children and groups of teachers. The manual has been written so that selected chapters can be used directly with 4th or 5th form groups taking a short appreciation course. Other chapters are designed more for the specialist taking one of the new 'A' level examinations in computer science. No advanced mathematical knowledge is assumed but it is expected that students using the manual will probably have followed one of the 'modern' 'O' level courses. It is particularly useful for would-be students to be familiar with binary arithmetic, flowcharting and elementary Boolean algebra.

I would like to acknowledge the help given by Benedict Nixon who first demonstrated that computer science could be devoid of the usual jargon, and Professor Bryan Higman for many useful suggestions. I should also like to thank Mrs. Sydney Hassall for trying out the new ideas and providing so much enthusiastic support and Mrs. Jenny Aellen without whose programming assistance and debugging ability the system would never have been written. Finally I should like to thank the Polytechnic for its financial support.

W. Tagg,
July, 1969.

CONTENTS

Chapter 1 Introduction

The computer described here is a hypothetical or 'paper' computer. Although it does not exist, there is no reason why a computer should not be built to this specification. An Elliott 803 computer at The Hatfield Polytechnic has been programmed so that it will behave exactly like the hypothetical machine; this manual has been written for users of this implementation. The design of the machine has had educational requirements in mind and for this reason it differs from a real machine in a number of respects. Most of these differences are unimportant and are no greater than the differences that exist between one real machine and another. In a later chapter we shall consider some of those differences in detail but for the moment we shall concentrate on one particular machine and learn to use that properly. One other aspect of the BBC machine which needs mentioning here is its capability of dealing with more than one user simultaneously. Computers that do this, must not make all of their facilities available to the ordinary user; there has to be some protection so that one user's program is not able to interfere with another user's. When reading the next few chapters, therefore, it should be borne in mind that there is more to the computer than the part **the user has access to.** In particular, there are built in routines which are automatically entered when a user attempts to do something which is clearly mistaken.

Input and Output

There must always be some method of passing information from user to machine and from machine to user. In our case, an on-line teleprinter is normally used for this purpose. 'On-line' means that there is a wire, capable of transmitting information, permanently connecting the teleprinter to the computer. When the user types on the teleprinter, the printed copy he obtains is only given for his benefit, what really matters is the electric signals which are sent to the computer. Electric signals pass in the opposite direction when the computer is to output information and these signals cause corresponding messages to be automatically typed for the user to see.

Input to the computer is via one of two languages. The computer has already been programmed so that it is capable of 'understanding' these but it must be remembered that the computer behaves like a complete moron and an error will result unless the languages are used precisely. The two languages are called the command language and the assembly code.

The Command Language

This is a high level language (i.e. it is a long way removed from the natural language of the computer itself) and is used to control the computer in a general way. Instructions given to the computer via the command language cause the computer to take some action immediately. The method that the computer uses is not explained, the user sees only the results it produces. Altogether, there are over twenty different commands and each consists of a single word surrounded by two apostrophes, thus:-

'RUN' .

Sometimes the command is followed by an expression, word or number in which case it might appear like:-

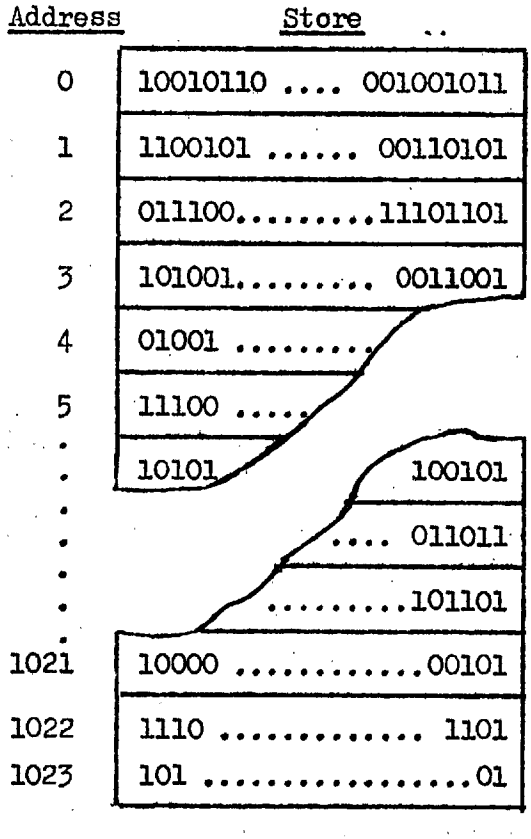
'LOGIN' BLOGGS

The Assembly Code

This is associated with the actual instructions which the computer is built to understand (called machine code). Before explaining the details, it is necessary to learn a little more about the computer itself.

The Store

One of the most important parts of a computer is its store. Basically, this consists of many thousands of binary digits (called bits) each of which is capable of taking one or other of the two values 0 and 1. These bits are grouped together to form 'cells'*; each cell has its own address by which it is known:



The store is used to hold information, all of which must be coded in some way so that it can be expressed in 0's and 1's.

It is fairly easy to see that numbers like 134 can be expressed in binary so that

0000 00010000110	could represent 134,

* Sometimes called 'words' or 'registers'

but much of the information that a computer handles is not easily thought of as a series of positive integers so that different methods are used for storing different kinds of information.

One of the chief kinds of information which a computer store must hold is the set of machine code instructions which make up a program. Most of the time, a computer works by obeying instructions which are already in store. This implies that two stages are involved if we want the computer to do a particular job. Stage 1 consists of putting the instructions to be obeyed in store and stage 2 consists of obeying them.

Examples 1

1.1 One way of coding information which can easily be expressed in alphabetic form is to code each letter into binary. In the BBC, it is possible to pack 4 letters (or other characters) into each cell. Each character is represented by 6 bits and the code used for this is fairly obvious. What message has been stored here in cells 29 to 34. (It might help to know that 111 101 represents a space.)

29	11	001 101	000 001	010 010	011 001
30	11	111 101	001 000	000 001	000 100
31	11	111 101	000 001	111 101	001 100
32	11	001 001	010 100	010 100	001 100
33	11	000 101	111 101	001 100	000 001
34	11	001 101	000 010	000 000	000 000

these bits tell the computer that a group of up to 4 letters (or other characters) are stored

these lines do not exist in the computer, they are put here to help you.

1.2 Using 6 bits to code each character, how many different characters can be coded? (In this context, a space is regarded as a character, but 000 000 is used to represent 'no character'.)

1.3 In 1.1 it was seen that the two left hand bits (called type bits) were used to specify how the other 24 bits were to be regarded. How many different types can be described using 2 bits? What other types would you think it might be useful to have?

Chapter 2 The First Program

```
16  TAKE  19
17  ADD   20
18  PUT   21
```

This (very short) program consists of three instructions which are to be stored in cells 16, 17 and 18. The cells with addresses from 0 to 15 have special properties and are not usually used for storing instructions so that the first instruction in a program often goes into cell 16. One of the special cells (with address 1) is called an accumulator and it is this cell which is used to store one of the two numbers (or other store content) which the computer is to process. The instruction TAKE 19 means:

> 'Throw away the current content of the accumulator (cell 1) and replace it with a copy of the content of cell 19.'

ADD 20 means

> 'Add a copy of the number in cell 20 to the number in the accumulator and leave the answer in the accumulator.'

This instruction implies that the current content of cell 20 and the current content of the accumulator are both to be considered as numbers (for the moment, whole numbers).

PUT 21 means

> 'Throw away the current content of cell 21 and replace it with a copy of the content of the accumulator.'

It is important to notice that TAKE and PUT both mean 'copy'. The original, which is being copied, is left intact.

Notation In future we shall write 'C( )' for 'content of' 'A' for 'the accumulator' and ':=' for 'becomes a copy of'. Using this notation:

TAKE 19	means	C(A) := C(19)
ADD 20	means	C(A) := C(A) + C(20)
PUT 21	means	C(21) := C(A)

The program in cells 16, 17 and 18 is not much use as it stands because so far we have not said anything about the content of cells 19 and 20. Suppose we arrange for C(19) to be +153 and C(20) to be +231. We could do this by writing the program as:

```
16  TAKE  19
17  ADD   20
18  PUT   21
19  +153
20  +231
```

This way, the numbers +153 and +231 are translated into binary by the assembler along with the instructions TAKE, ADD and PUT, but we still have difficulty because although the computer is capable of adding the two numbers and putting the answer into cell 21, we have not included any method of finding out what that answer is. For the moment, we shall use the command 'DUMP' for this purpose.

'DUMP' is always followed by a pair of addresses and causes the current content of each cell within the range specified by these addresses to be typed on the teletype. (If the content of a cell is all zeros, the cell's content is not output.)

Let us now look at the complete conversation with the computer. It is important to realise that the computer deals with a whole line of information at once and responds by typing either a question mark or apostrophe when it is the user's turn to type. The lines which have been typed by the computer are marked thus: [

```
 'LOGIN' BLOGGS          Note: 1
[BLOS                    Note: 2
 ? 16 TAKE 19            Note: 3
 ? 17 ADD  20
 ? 18 PUT  21
 ? 19 +153
 ? 20 +231
 ? 'RUN'                 Note: 4
[ERROR  No. 17           NO INSTR. IN THIS CELL  L = 19   Note: 5
 'DUMP' 16, 21           Note: 6
[16 TAKE 19
 17 ADD  20
 18 PUT  21
 19 +153
 20 +231
 21 +384                 Note: 7
```

Note 1 Each user must start with 'LOGIN' followed by his name code. The computer has been warned what names to expect and will have nothing to do with you if you use a different name. The computer needs to know who you are so that it can find certain records associated with you. In particular, the computer makes a note of how many mistakes you make!

A completely separate reason for the 'LOGIN' sequence is to arrange for the new user to start with a 'clean sheet'; the previous person's work is removed from the store.

Note 2 When the 'LOGIN' sequence is complete (it takes a few seconds), the user's name is typed back as confirmation. Notice that only the first three and last characters of the name are recorded, the user could have successfully logged in using the name

BLOTHISISAVERYLONGNAMEGGS
or
BLOS

Note 3 The user types the address of a cell followed by its required content. A question mark is output by the computer when it is ready for the next line.

Note 4 So far, the computer has spent its time translating instructions and numbers into binary; it has not done any useful work. 'RUN' is an instruction to the computer telling it to start obeying the user's stored instructions in sequence starting with the first one that was typed in.

Note 5 The computer obeyed the instructions in cells 16, 17 and 18 and then attempted to obey the 'instruction' in cell 19. Since cell 19 did not contain an instruction, the program was brought to a halt and a warning was output. Later, we shall see how we can stop a program rather more elegantly.

Note 6 Output (unless zero) the current content of cells 16 to 21 inclusive.

Note 7 The Answer.

Examples 2

2.1 During the logging in sequence, why did the computer only bother with 4 letters out of the user's name?

2.2 How was the computer able to detect that C(19) was not an instruction?

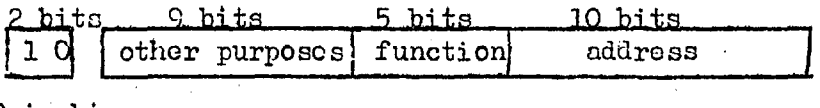
2.3 Here are two more instructions:

MPLY 27 means C(A) := C(A)*C(27)
(we shall use * to mean 'multiply' since it cannot be mistaken for a letter)
SUBT 53 means C(A) := C(A) - C(53).

Using instructions like these (together with instructions like those already used), write a program to work out the value of 14*27 - 18*16.

Chapter 3 The Instruction Format

The 24 bits that make up an instruction are grouped into several fields. Each field tells us something about the instruction:



1 0 in binary specifies an instruction

The ten bits at the right hand side of the cell normally specify the address used in the instruction and the next 5 specify one of 32 functions or operations which can be performed. So far we have used 5 of these functions viz:

TAKE PUT ADD SUBT MPLY

Each of them causes various pieces of electronic circuitry to be brought into play so that the computer can carry out the appropriate computation. A computer is severely limited because it can only carry out the operations specified by its repertoire of functions and these tend to be fairly basic things like adding and copying. Because users want to do less basic things, the assembler for the BBC machine (in common with most assemblers) besides taking on the responsibility of translating instructions into binary, provides the user with a number of Library routines which carry out the tasks not supplied in the basic machine.

These routines are written in the basic machine code within the assembler itself and occupy a separate area of store which is inaccessible to the user in the normal way. As it happens, none of these routines require more than a single operand, (an 'operand' is C(a store location) on which a particular operation is performed) so that information is passed to and from the library routines via the accumulator.

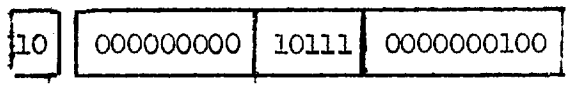
To start with we shall introduce three library routines called READ, PRINT and STOP. They are defined as follows:

READ - Take, from the teleprinter at RUN time, a store content and place it in the accumulator. For the moment, this store content will be representing an integer and the READ routine will have the task of translating this integer into binary.

PRINT - Print out, on the teleprinter, the current content of the accumulator. For an integer this implies binary to decimal conversion. After a PRINT instruction, the content of the accumulator remains unchanged.

STOP - Print out an END OF PROGRAM message and get ready to read the next command.

All these library routines can be called by writing LIBR followed by a number to specify which particular routine is required. This number is stored in the address field of the instruction but it is no longer an address; it simply specifies which routine is required. An alternative way of specifying a library routine is to type its mnemonic so that LIBR 4 and READ are both acceptable to the assembler and both get translated into:



(10111 is the particular binary pattern corresponding to LIBR).

Let us now look again at the program given in Chapter 2. Using READ and PRINT the complete conversation might look like this:

```
'LOGIN' BLOGGS
[BLOS
 ? 16 READ
 ? 17 PUT 100
 ? 18 READ
 ? 19 ADD 100
 ? 20 PRINT
 ? 21 STOP
 ? 'RUN'
[DATA?                          Note: 2
 153, 231
[384
[END OF PROGRAM
 'RUN'
[DATA?
 27, 191
[218
[END OF PROGRAM
```

Note: 1

Note: 3

Note 1 Line by line storage of instructions.

Note 2 When a program comes to a READ instruction (at RUN time, not when it is being translated), DATA? is output and the user is invited to type up to 1 line of data or information for the program to process. If the user supplies less than the required amount of data the program will stop running when it has run out, and DATA? will be output a second time. If the user supplies too much data the program is terminated with

UNUSED DATA:-

followed by the next 6 characters of unused data (or less if there are less than 6 characters left).

Note 3 This program has been run twice with different sets of data.

Examples 3

3.1 The addresses of the cells available to a user go from 0 to 1023. What is special about 1023? Give two reasons why a limit is imposed.

3.2 Write, using as few instructions as possible, a program which will read in a number, find its sixth power and print out the result. Bring your program to a satisfactory halt.

Chapter 4 Correcting Errors

The rubout character

Input to the computer is organised on a line by line basis. Until a line is completed, the computer takes no action other than to store away the characters in a special area ready for future processing. The new line characters terminate this process but until these have been typed, it is possible to remove the last character that has been typed by typing an &. This character is not stored; it simply causes the previous character to be removed. && removes two characters and so on but if more &s than characters have been typed on a line, the extra ones are ignored. Only 67 characters can be input on one line, more than this is detected as an error and the whole line is cancelled.

The commands 'EDIT' and 'ADVANCE'

When a program is being input, the assembler checks to make sure that consecutive cells are being used and a warning is output if the user fails to comply with this requirement. Often, mistakes in a program are only discovered after the program has been input. The command 'EDIT' is a directive to the assembler saying that until further notice, instructions may be input in any order. Wrong instructions can be overwritten by new instructions or just cancelled by overwriting with a 'do nothing' instruction (written as NTHG 0). The computer always indicates when it is the user's turn to type by suitable output. This output consists of either a question mark (inviting an instruction), an apostrophe (inviting a command) or DATA? (inviting the user to type up to one line of data). A command can always come in place of an instruction and often in place of data but the reverse is not true.

'ADVANCE' enables the user to move part of a program about the store so that extra instructions can be inserted. It is followed by 3 integers which specify the details of the required change. The first of these (which must have a sign) specifies the number of places through which part of the program must be moved and the other two integers represent the first and last addresses of the section of program which is to be relocated. 'ADVANCE' + 3, 23, 37 means move the section of program stored in 23-37 3 places so that it occupies the cells with addresses 26-40. 'ADVANCE' - 3, 23, 37 moves the section 23-37 to 20-34.

'LIST' and 'DUMP'

When a program has been edited a number of times, a user may require a fair copy. 'LIST' on a line by itself will cause the computer to output a complete copy together with the necessary commands for it to be successfully re-input. Certain helpful comments (given in brackets) are included but non-zero store contents (e.g. data areas) are not output. The computer sometimes has difficulty in distinguishing program from data, indeed there is no hard and fast distinction but for most purposes the computer has little difficulty provided certain conventions are kept to.

By comparison 'DUMP' is not concerned with interpretation at all. The contents of all locations in the specified range are output (unless zero).

To summarise, 'LIST' 23, 44 gives a list of the program segment stored in cells 23-44 while 'LIST' on a line by itself gives the whole program. 'DUMP' 23, 44 gives the non zero store content of any location between 23 and 44 (inclusive) and 'DUMP' on a line by itself is not allowed.

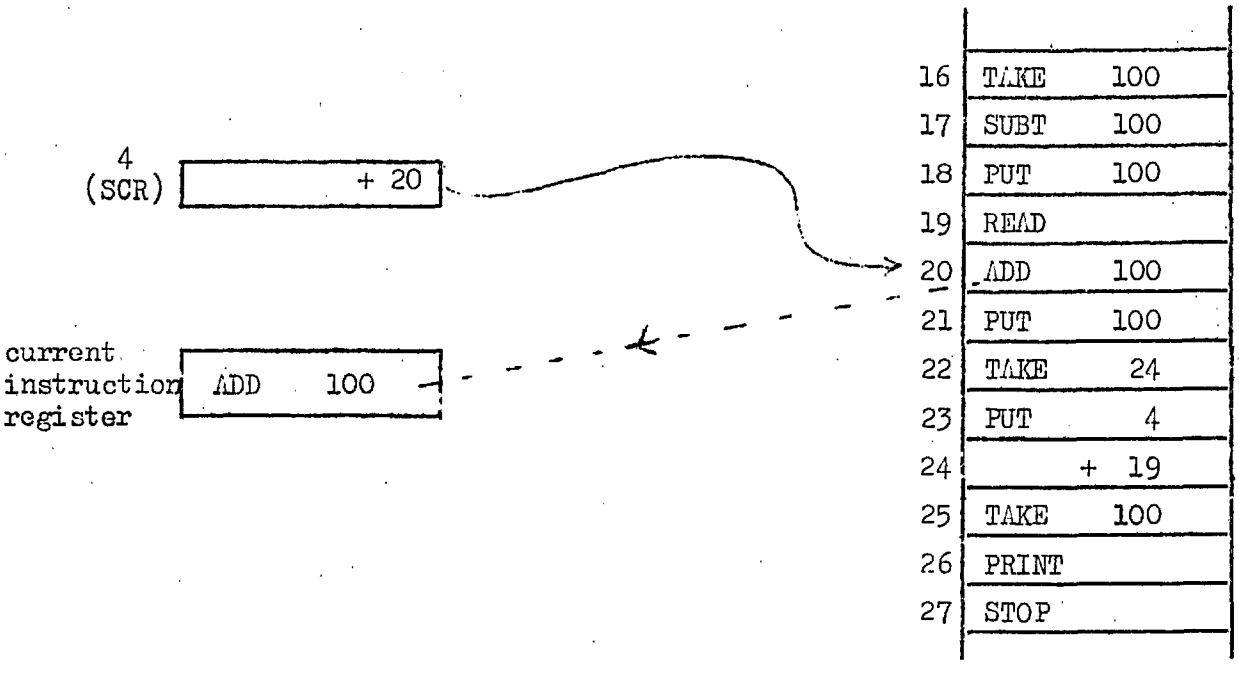
Examples 4

4.1 Write out a complete copy of what you think the output would have been if the worked example in chapter 3 had continued with the command 'ADVANCE' + 78, 20, 21 followed by 'DUMP' 95, 100.

4.2 If you have access to a BBC terminal, try out these various commands even if you have had no occasion to use them so far. In practice, programs seldom function correctly first time; often, it is necessary to edit a number of times to achieve perfection.

Chapter 5 Jump Instructions

One of the chief reasons for storing instructions rather than obeying them immediately is so that they can be obeyed a number of times by incorporating them in a loop. Up till now, little has been said about the order in which instructions are obeyed, but in fact a complete sequence of events takes place inside the computer each time an instruction is obeyed. There is a special cell (with address 4) which has the task of "keeping the computer's place" by storing the address of the current instruction. Cell 4 is called the sequence control register and it is automatically increased by 1 each time through the cycle of events associated with obeying one instruction.



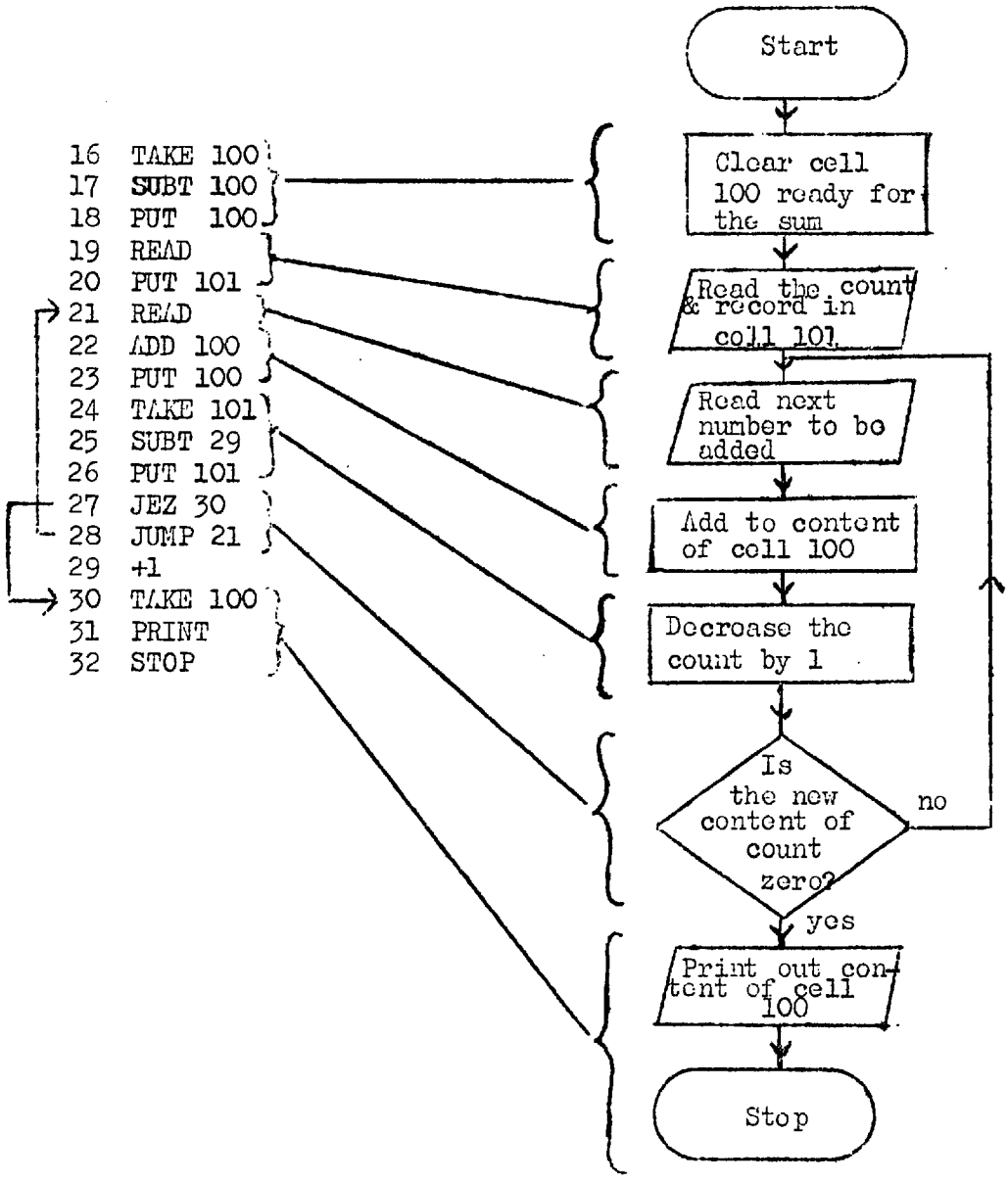
'RUN' on a line by itself is interpreted by the computer to mean run from the first instruction and causes the sequence control register to be loaded with +16. The computer uses the SCR to access the cell it points to and a copy of the content of this cell is taken to another special cell (not addressable) called the current instruction register and it is from here that it is obeyed. The exact point at which the SCR is increased by 1 is important. It varies from machine to machine but for our machine it comes after the next instruction has been copied into the current instruction register but before it has been obeyed.

A study of the program given here will show the reader how these facts can be used to cause the instructions in cells 19-23 to be obeyed again and again. The program adds together a series of integers and records the answer in cell 100. The first three instructions cause zero to be placed in cell 100 (in rather a clumsy way). Instructions 19-21 read a number from the data and cause it to be added into the current content of 100. Instructions 22-23 cause the SCR to be set to 19 so that the instruction in cell 19 is obeyed next. This sequence of events can only be broken by a command in place of data and in this case 'RUN' 25 is appropriate. When 'RUN' is followed by a space and an address, the SCR is loaded with this address instead of the address of the first instruction.

This method of closing a loop illustrates the problem but it is a little complicated. To simplify the procedure, a new function, JUMP is introduced. In this case, instructions 22-23 can be replaced by the single instruction JUMP 19 which causes the SCR to be reset to 19 without using the accumulator.

Even this method requires a command in place of data to break the loop so that we require a more sophisticated technique still. JEZ stands for Jump equal to zero. If the accumulator is zero a Jump takes place, otherwise the instruction is ignored and the computer goes on to obey the next instruction in sequence. To use this instruction effectively it is necessary to incorporate a sequence of instructions to count the

number of times round the loop. Suppose that the numbers on the data are preceded by a number specifying how many numbers are to be added. The following program will obey the loop the requisite number of times and then exit automatically.



Here are three more functions:-

JNZ	(Jump not zero)	Similar to JEZ. Jump if the accumulator is not zero, otherwise do nothing so that the next instruction in sequence is obeyed.
DECR	(Decrease by one)	Decrease the content of the store named by one and replace the content of the accumulator by a copy of the old content of the named store.
INCR	(Increase by one)	Increase the content of the store named by one and leave the content of the accumulator unchanged.

Using the notation introduced in chapter 2 we may rewrite these definitions as follows, provided we take note of the order in which the changes are to be made:-

JNZ 100 means if C(A) $\neq$ 0 then NTHG 100
if C(A) $\neq$ 0 then C(4):= +100

DECR 95 means C(A):= C(95), C(95):= C(95) - 1

INCR 89 means C(89):= C(89) + 1 **and C(A) is unchanged**

Here is a cell with a special property:-

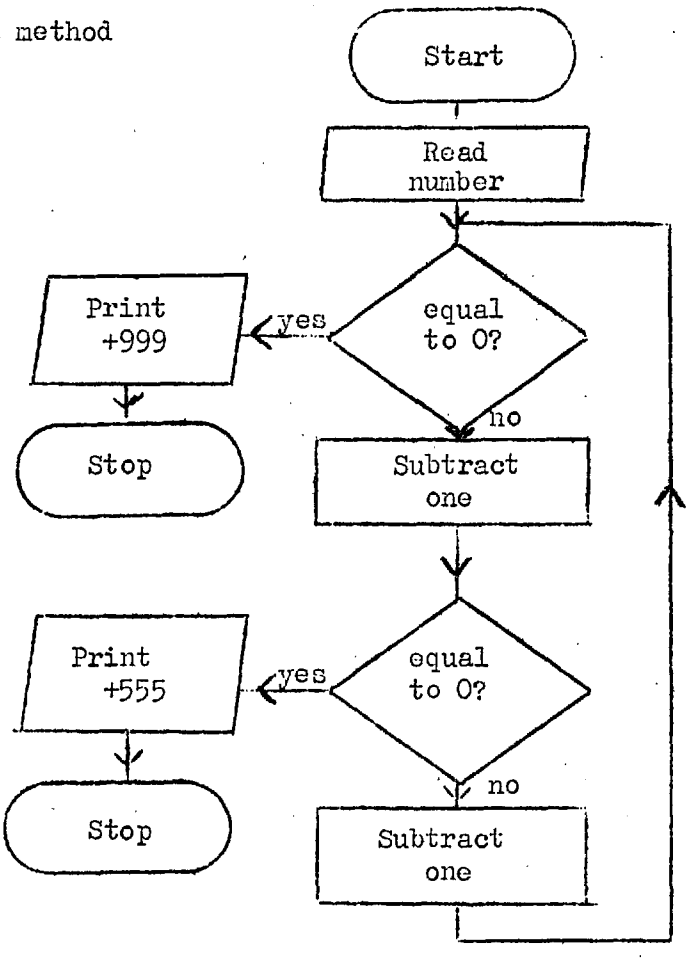
Cell 0 has the special property of being incapable of storing anything. It always appears to be the source of zero. TAKE 0 is particularly useful since it is guaranteed to cause C(A):= 0

Examples 5

5.1 Rewrite the last example using JNZ, DECR and TAKE 0.

5.2 The flow diagram shows a method of deciding whether or not a number is even. If it is even + 999 is printed out, and if it is odd, + 555 is printed out.

Write a program using this method, but do not test it with too large a number. It is a very inefficient method.



Chapter 6 Backing Store

Programs, half completed programs or programs that are only half way through their computation, can be filed away so that more work can be done with them on another occasion. This is done by copying the whole of a user's store onto another storage medium. In the case of the BBC, the storage medium is magnetic film; it works rather like a tape recorder and before information can be stored or retrieved, it is necessary for the film to turn on large spools so that the appropriate section is under the read/write head. The time taken for this search to take place varies according to its initial position, but if it is badly placed it could take some minutes.

At The Hatfield Polytechnic, there are 3 of these film handlers and each user is given enough room to store a complete copy of his main store on each handler. He controls this transfer by using the commands 'FILE' and 'FETCH'. Each command is followed by a 1, a 2 or a 3 to specify the handler, but it is not necessary to specify the particular area; the computer allocates this automatically using the user's name to identify the required section.

'FILE' 1 means: search for the appropriate area on handler 1 and copy the current user's store content into that area, wiping out the existing information stored in this area to do so. The user's main store content remains unchanged.

'FETCH' 2 means: search for the appropriate area on handler 2 and copy the content of this area into the main store. Again, it is a copying process which leaves the original unchanged.

Because the computer is only capable of undertaking one search at a time (it can do other things while this search is taking place), the message FILM BUSY may sometimes be displayed. In this case a user must either wait and try later or do something else.

Each film handler holds enough film for 256 users. Some of these users are working from other outstations, but other areas of the film have been reserved for programs that are already written and working. These are made available to the user by name, if 'FETCH' is followed by STNDV for instance, a program to evaluate standard deviation is brought into the main store. A user must type 'RUN' 16 to find out how to use such a program. Instead of running the program, this will cause explanation (stored with the program) to be output. To run the program, 'RUN' on a line by itself should be typed.

'FILE' STNDV is not possible, but of course there is nothing to stop a user obtaining a copy of one of these system programs and either filing it in his own area, altering it or listing it.

Before a program is filed, the complete content of the user's store is added up and the grand total is stored with the program. This total is compared with a second grand total computed when the command 'FETCH' is used. These two totals, of course, will normally agree, but their failure to do so indicates that either the transfer has not been successful (unlikely) or that the information stored on the file has deteriorated. In such a case, the message SUMCHECK FAILURE is displayed. Under such circumstances, the file is lost. The message FILM FAILURE is caused by a failure to set up the film handlers correctly. If the message persists, the centre should be informed.

Interrupts When the computer is listing, dumping, searching for or running a program, it can be interrupted by typing any key. This is only effective when the computer is not in the middle of output so that when a program is being listed, it should be interrupted during the brief pause between lines. After an interrupt, the computer expects a command; a program that was interrupted while it was running can be made to continue from where it left off (even after alterations have been made or it has been filed and retrieved) by typing the command 'CONTINUE'.

Chapter 7 More Help from the Assembler

So far, the work done with BBC has been described in terms of the computer itself rather than from the point of view of the user. No programming language would be worth much unless it was organised so that it was easy for use. In the last few years, great strides have been made in this area and modern programming languages have become very sophisticated. In this chapter, we introduce two ideas which, in themselves add nothing new; they are only concerned with making the assembly language more readable. Consider the following problem:

A program is to be written which will compute a total mark for each pupil in a class by adding his English mark to 3 times his Maths mark. We are concerned with this small piece of the total program. We may write

TOTAL = ENGLISH + 3*MATHS.

and this can be coded for the computer using cell 100 for the total, 101 for the English mark, 102 for the Maths mark and 103 for the constant +3. We should write something like:

```
196 .........
197 TAKE  103
198 MPLY  102
199 ADD   101
200 PUT   100
201 .........
etc.
```

and we should have to remember which address was being used for which purpose. There is no reason, however, why the computer should not be given the task of deciding which cell to use for which purpose. As long as it is consistent, it makes no difference where these items are stored. Normally, this is precisely what happens and the above segment of program is written:

```
196 .........
197 TAKE+3
198 MPLY MATHS
199 ADD ENGLISH
200 PUT TOTAL
201 .........
```

It is usual to warn the computer at the head of the program (or elsewhere) that certain words are going to be used instead of addresses. This is done by using the command 'DECLARE' followed by a list of the words to be used. In this case we should write:

'DECLARE' MATHS, ENGLISH, TOTAL.

Any combination of letters and decimal digits can be used to name a cell in this way provided it starts with a letter. Such names are termed 'identifiers'; the following are legal:

A1 B27 A2B3 TWICE and the following are illegal:
1B (does not start with a letter) AB*C (contains an asterisk).

When an identifier is declared, a pair of cells are reserved at the top end of the store. One is used to store the name of the identifier itself, and the other is used as work space for that identifier and its address is assembled into the program. In Chapter 1 it was learned that a cell could hold up to 4 characters but some of the identifier names we have been using are longer than this. The computor gets over this problem by only storing the first three and last characters of the identifiers. Because of this, it is important to choose words that are distinct in this respect; the computer would regard STRING and STRIKING as the same. Normally, if a user forgets to declare an identifier, a warning is output the first time he uses it, but the assembler will automatically declare it for him, so that it is usually an error he does not need to correct.

The assembler uses a similar technique to cope with the instruction TAKE+3. The constant +3 is stored in a cell at the top end of store and the address of this cell is assembled into the instruction. It is important to notice the precise way in which this instruction is typed. Normally, the address following the function is separated from it by at least one space. When the address is replaced by a constant this space must be replaced by a + or - sign; it is a mistake to include the space as well. Clearly, only certain functions are appropriate for use with a constant. PUT+3 does not make sense and is not allowed.

Examples 7

7.1 A sequence of numbers with interesting mathematical properties known as the Fibonacci series is to be printed out with 8 members of the sequence to each line. Part of the sequence is shown below; apart from the first two terms, which are both 1, each term is formed by adding the two preceding terms.

1, 1, 2, 3, 5, 8, 13, 21 .....

A 'new line' can be output with the results by using the Library routine LINE. The flow diagram for the process is given below, and it is suggested that the user should declare the following identifiers:

COUNT - to count how many numbers are printed to a line

NEXT - the next Fibonacci number

LAST - the last Fibonacci number evaluated

LTB1 - the last but one Fibonacci number evaluated.

A possible start to the program is as follows:

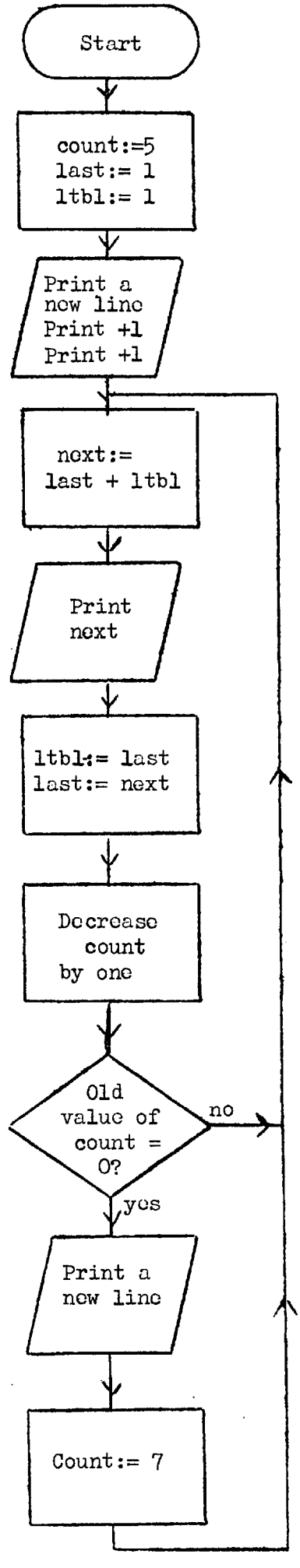
```
'LOGIN' SMITH
SMIH
? 'DECLARE' COUNT, NEXT, LAST, LTB1
? 16 TAKE+5
? 17 PUT COUNT
? 18 TAKE+1
? 19 PUT LAST
? 20 PUT LTB1
? 21 LINE
? 22 PRINT
? 23 PRINT
    etc.
```

Notice that we write:
last:= next
when we want to say 'content of the cell called LAST is replaced by a copy of the content of NEXT'.

This notation is often more convenient than the notation introduced in chapter 2. In future, lower case letters will be used to signify 'content of' so that:

next
and
C(NEXT)

mean the same thing.



When the program is running, it will be necessary to interrupt it when sufficient output is obtained.

At this stage, it would help to get a complete 'LIST' of your program so that you could see how space at the top end of store had been allocated.

Chapter 8 Different types of Storage

The BBC cells are capable of storing different kinds of information. Provision is made for four basic types but in fact there are many more than this; the content of a cell is only a series of 0's and 1's, meaning is a question of the interpretation that is placed on these. The four explicit types we shall describe, are important because they cause the various functions to operate on them in different ways.

Associated with each cell, two special bits (called the type bits) describe between them the manner in which the other 24 bits are to be interpreted.

The four types are:

type bits	name	meaning
0 0	I-word	Integers (whole numbers), positive or negative. (Numbers outside the range -2^{23} to $+2^{23}-1$ can not be stored as I-words)
0 1	F-word	Floating-point numbers. This is a more general way of representing numbers which may be very large or may be fractional. There are limitations on accuracy.
1 0	P-word	Program words or instructions.
1 1	S-word	Strings of up to four characters.

Before going on to describe in detail the ways in which information of these various types is stored, it might be useful to describe, from the user's point of view, how these cell contents are treated by a program. Table 1 in the appendix shows that certain functions are inappropriate for certain types of store content; the first thing that the user must learn is that a program which is nonsense will either produce an error indication or a nonsense answer. Let us consider the TAKE instruction. Essentially, this is concerned with copying; it makes sense to copy anything - even the instructions of the program itself. The sequence of instructions

```
16 TAKE 16
17 PRINT
   etc.
```

may seem a little pointless but they have a meaning so that when they are obeyed

```
TAKE 16
```

is output. By contrast,

```
16 TAKE 16
17 ADD 20
   etc.
```

is meaningless. An instruction is not an arithmetic concept so that ADD is inappropriate. An attempt to run such a sequence of instructions will result in the error

```
OPERAND OF WRONG TYPE
```

being displayed.

Common sense will enable the user to decide most of the time whether an instruction sequence makes sense, those interested in the minute details are referred to the appendix or to the four next chapters (i.e. 9 to 12). For those who are less inquisitive the details given in the rest of this chapter are probably sufficient.

Because they are really pieces of program in their own right, the Library routines READ and PRINT are particularly versatile. PRINT causes an instruction to be output in its mnemonic form, an S-word to be output just in the form of the characters that form it while F-words and I-words are merely printed in terms of their equivalent to base 10. F-words normally contain a decimal point and are terminated by one space while I-words are terminated by two spaces. Normally, room is allowed for 5 digits and a sign (but the number of digits can be altered) so that altogether 8 characters are output.

READ is a more difficult routine to implement. If 3427 is met as data during a READ instruction, it is not easy for the computer to determine whether this is to be regarded as the integer three thousand, four hundred and twenty-seven or as the character string composed of the four characters 3, 4, 2 and 7. To solve this problem, the computer first inspects a special cell (with address 11). If C(11) = 1 then the next 4 characters are read and stored as an S-word, otherwise some sort of number is expected. The presence of a decimal point (or subscript 10) designates an F-word, otherwise the number is stored as an I-word. Two special characters are used to control C(11). These are the quote and unquote characters (< and > are used). During a READ instruction, a quote sets C(11) to +1 but is otherwise ignored, whilst an unquote sets C(11) to 0. The usual way to input an S-word therefore is to surround it with a quote and unquote thus:

< AB* >

A READ instruction would send such data to the accumulator thus:

type bits	A	B	*	
11	000001	000010	110010	000000

For the specialist, octal (base 8) input is possible (see appendix), otherwise P-words can not be input as data to the user's program using a READ instruction.

There remains one other useful library routine which should be mentioned at this point. This is called by the mnemonic CAPTN (for caption). It takes up to 11 S-words as operands and causes them to be output in order to form a message. These S-words are placed in successive cells immediately following the CAPTN instruction and the sequence is terminated by an ordinary instruction. An example will make this clear:

```
.....
.....
26 TAKE  ANS
27 LINE
28 CAPTN
29 <THE >
30 <ANSW>
31 <ER= >
32 PRINT
.....
```

This sequence of instructions, if obeyed, would cause THE ANSWER = to be output on a new line followed by the current content of the accumulator. CAPTN does not alter C(A) but it does alter C(SCR) so that (in this case) the instruction in cell 32 can be obeyed immediately after the instruction in cell 28.

Examples 8

8.1 The instruction DVD when used with two I-words causes the C(A) to be replaced by another I-word which represents the result of division (i.e. C(A) ÷ C(store)). The positive remainder after division has taken place is stored in the remainder cell (with address 3). Incorporate this instruction into a program that will read a number of pence from the data and convert it to pounds shillings and pence. Typical output should be of the form:

326 PENCE = L 1 - 7 S - 2 D

8.2 Re-run some of your earlier programs and use F-words as data. It would be helpful to re-run 3.2 with the following sets of data:

(i) 2 (ii) 2.0 (iii) 1.7321 (iv) 173.21

(the last answer will be a little unexpected).

Chapter 9 I-words

When integers are to be stored in the computer, coding them as their binary equivalent is sufficient provided that the integers are small enough and are positive. If the integers are too big, a special cell (called the overflow cell) records the fact. We shall deal with the problem of overflow in a more detailed way at a later stage. All the user needs to know at the moment is that the message INTEGER OVERFLOW will be displayed if an attempt is made to store an integer which is too large to fit into a single cell.

Negative integers are stored by using a convention known as two's complement. This technique is best illustrated by using a smaller word length. We shall use a four bit cell:

Without using 2's complement		With the use of 2's complement	
binary (base 2)	denary (base 10)	binary (base 2)	denary (base 10)
0000	0	0 000	0
0001	+1	0 001	+1
0010	+2	0 010	+2
0011	+3	0 011	+3
0100	+4	0 100	+4
0101	+5	0 101	+5
0110	+6	0 110	+6
0111	+7	0 111	+7
1000	+8	1 000	-8
1001	+9	1 001	-7
1010	+10	1 010	-6
1011	+11	1 011	-5
1100	+12	1 100	-4
1101	+13	1 101	-3
1110	+14	1 110	-2
1111	+15	1 111	-1

When 2's complement notation is used the most significant bit is reserved as a sign bit. If it is a 0, the rest of the bits describe an integer which is either positive or zero, but if it is a 1, the rest of the bits describe a negative integer. The following list illustrates how a few typical integers are stored in the BBC.

type	sign		
00	0	11111111111111111111111	+ 8 388 607 or $2^{23}-1$. The largest integer it is possible to store.
00	1	00000000000000000000000	- 8 388 608 or -2^{23}. The smallest integer it is possible to store.
00	0	00000000000000000001110	14
00	1	11111111111111111110010	-14
00	1	11111111111111111111111	-1

Examples 9

9.1 Write out the integers from -16 to 15 and against each, write their binary representation using 2's complements and a five bit word. Using these, verify that the results of the following are correct (provided spare bits are allowed to 'drop' off the end) if the usual methods for handling binary are observed.

(i) 4 + 7 (ii) -9 + 6 (iii) 14 + (-5) (iv) -4 + (-7)

Show that the following do not yield correct results and explain why:

(v) -7 + (-10) (vi) 8 + 9.

9.2 In the BBC order code, two functions are available which allow a program to branch according to the sign of the accumulator.
JLZ means 'jump if the accumulator is less than zero, otherwise obey the next instruction in sequence'.

JGZ means 'jump if the accumulator is greater than zero , **otherwise obey the next instruction in sequence.**'

Data consists of a series of positive and negative integers terminating with a zero. Write a program to count and print out the number of positive integers followed by the number of negative integers.

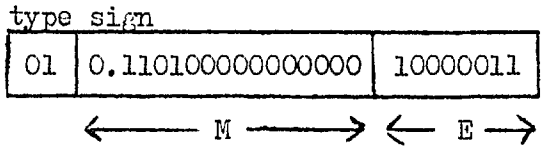
9.3 Write a program which reads n followed by n integers, and prints out the largest and the smallest.

Chapter 10 Floating point representation

Calculations of a scientific nature for which computers are used, are normally concerned with handling data which does not consist entirely of whole numbers. Often, too, the numbers involved vary greatly in size even within a single calculation. Floating point representation is a method frequently used to store data of this kind. Although the method is complicated, it is very easy to use.

In the BBC, floating point representation is achieved by grouping the 24 bits available into two fields:

An F-word



The M field is used to store a binary fraction using 2's complement representation for negative fractions. The binary point is immediately to the right of the sign bit so that in the example given above M represents the fraction 0.11010... in binary or $\frac{1}{2} + \frac{1}{4} + \frac{0}{8} + \frac{1}{16} + \frac{0}{32} + \ldots$ in denary (base 10). The E field stores a positive integer in the range 0 to 255 and the content of the whole cell represents the number:

$$m \times 2^{e-128}$$

In the example above, C(E) represents 131 so that the complete cell stores

$$\frac{13}{16} \times 2^3 = 6.5$$

m and e are always adjusted so that the numerical value of m is always as large as possible, when m is positive there is always a 1 immediately on the right of the binary point and when m is negative there is always a 0 immediately on the right of the binary point. Because of this, the method of representing any number is unique, and zero is not capable of being represented at all.

Before going on to the examples, it is perhaps fair to point out that it is not necessary to have a deep understanding of this chapter to be able to follow the rest of the book. However, the following summary is important and should not be skipped.

Summary

(i) Floating point representation is a method used to store real numbers to an accuracy of 15 binary places (between 4 and 5 significant figures in denary).

(ii) Apart from nought (which must always be stored as an I-word) any number in the approximate range -10^{38} to $+10^{38}$ can be stored. Attempts to store numbers outside this range cause FLOATING POINT OVERFLOW to be displayed.

(iii) The functions ADD, SUBT, MPLY and DVD will accept I-words or F-words. In each case, the answer is an I-word if both operands are I-words and an F-word otherwise. READ and PRINT normally use the decimal point to distinguish the two types; it is nearly always present in an F-word and never in an I-word. The following examples illustrate this:

4 (I-word)	+	6 (I-word)	=	10 (I-word)
4.3 (F-word)	x	2 (I-word)	=	8.6 (F-word)
16.4 (F-word)	-	12.0 (F-word)	=	4.4 (F-word)

(iv) JLZ and JGZ depend on the sign bit of the accumulator for their action. For this reason, their action does not change if the content of the accumulator is an F-word. JEZ and JNZ depend for their action on the content of a special cell (with address 6) called the tolerance cell. If the E-field of the accumulator is less than or equal to C(6) then JEZ causes a jump and JNZ does not, otherwise JEZ does not cause a jump and JNZ does. C(6) is set to +16 when a user logs in, so that unless he changes this value, JEZ means (when C(A) is an F-word)

'Jump if the accumulator is very near to zero.'

and JNZ means

'Jump unless the accumulator is very near to zero.'

(v) INCR and DECR accept F-words or I-words.

Examples 10 (If you do not want to know how F-words are stored, skip 10.1 and 10.2.)

10.1 Represent the following numbers in floating point form by drawing a picture of a cell and its content.

(i) 4.75 (ii) -15.5 (iii) 256.0

10.2 What numbers do the following represent?

(i)

01	0.100000000000000	1000111

(ii)

01	0.111100000000000	01111100

(iii)

01	1.010000000000000	10000100

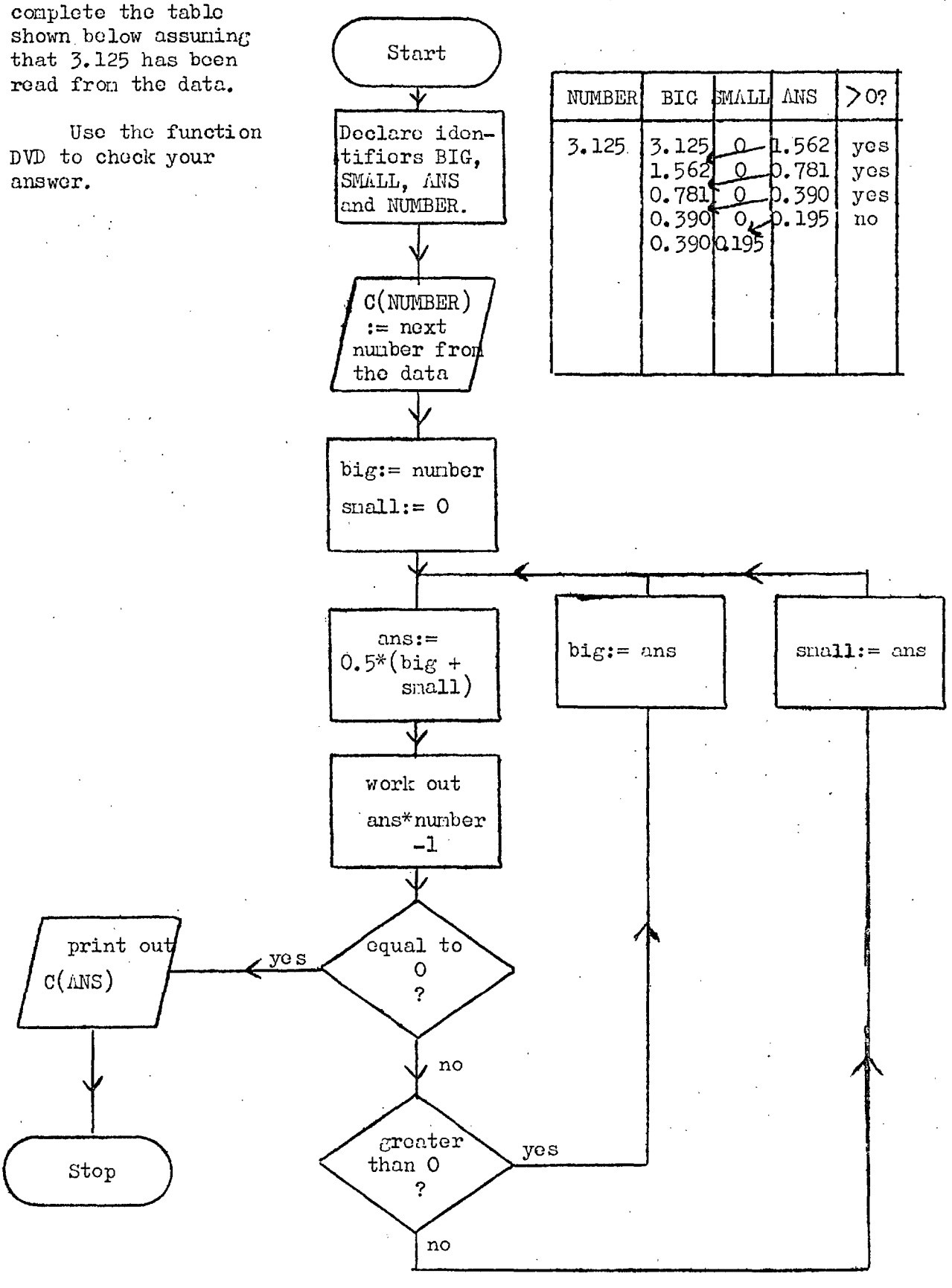
10.3 Re-run examples 9.2 and 9.3 using floating point data or a mixture of floating point and integer data.

10.4 Some very primitive computers do not have an instruction for performing division. In such a case, a user can manage if he can find the reciprocal of a number. **The method is called an iterative process, this is just a respectable name for organised guesswork.**

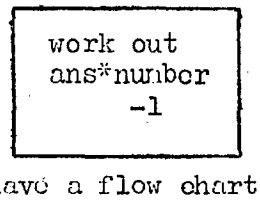
The flow diagram below illustrates a method of doing this provided the number is greater than 1. Before you write a program to do this complete the table shown below assuming that 3.125 has been read from the data.

Use the function DVD to check your answer.

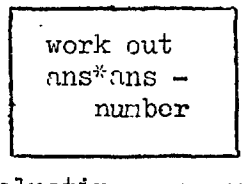
NUMBER	BIG	SMALL	ANS	>0?
3.125	3.125	0	1.562	yes
	1.562	0	0.781	yes
	0.781	0	0.390	yes
	0.390	0	0.195	no
	0.390	0.195		



10.5 Replace the box



with



in the above flow chart. We now have a flow chart for evaluating a square root. A program for evaluating the square root of a number can be checked by using the library routine SQRT.

Chapter 11 S-words and logic functions

Up to 4 characters can be coded and stored within a single cell. This is done by dividing the 24 bits into 4 groups of 6 bits each. Each of these groups describes a single character and the details of the code used are shown in Table 2 of the Appendix.

The basic instructions for getting characters into and out of a given cell are IPUT and OPUT. These are concerned with the transfer of characters into and out of the user's store; library routines like READ, PRINT and CAPTN are made up from these and other instructions.

IPUT 71 means take the next character from the data and store it at the least significant end of cell 71. The rest of the cell is cleared and the type bits are set to specify an S-word.

OPUT 153 means print on the teletype the character corresponding to the 6 least significant bits of C(153). C(153) must be either an I-word or an S-word.

These instructions are very basic but all other methods of getting information into or out of the student's store use them. If we wish to store more than one character in a cell, we need some way of moving characters (or other bit patterns) within the cell. In the BBC we have two instructions for doing this:-

SHFR 29 means shift the bit pattern within the accumulator to the right and feed in zeros at the other end. The number of places to be shifted is specified by C(29). C(29) must be an I-word in the range -24 to +24, negative values specifying a left shift. The type bits of the accumulator are not used and are not affected.

CYCR 53 means shift the bit pattern of the accumulator cyclically to the right. Cyclically means that bits that fall off one end are fed into the other end. C(53) again specifies the number of places.

It is clearly meaningless to add or multiply characters so the arithmetic instructions are invalid for character manipulation and other instructions are provided. The logic functions are the most convenient to use for this purpose. The distinguishing feature of these instructions is the fact that they cause a result to be formed on a bit by bit basis. There is no 'carry' as in arithmetic functions. To specify the result of a logical function, we only have to consider what has to happen to single bits and this process is duplicated for each of the 24 bits of a cell content.

Truth tables

a	b	a∧b
0	0	0
0	1	0
1	0	0
1	1	1

This table describes what happens if bits 'a' and 'b' are combined by a logical operation. In this case, we are told that the result is to be a 0 except in the case where a = 1 and b = 1. For this reason the operation or function is called by the mnemonic AND. The symbol ∧ is used to denote AND in the same way that + is used to denote ADD.

a	b	a∨b
0	0	0
0	1	1
1	0	1
1	1	1

This is the OR function because the result is a 1 if a = 1 or b = 1 (or both). The symbol ∨ is used to denote OR.

a	b	a≠b
0	0	0
0	1	1
1	0	1
1	1	0

The result this time is only a 1 if a and b take different values. This function is called by the mnemonic NEQV (which stands for not equivalent). The symbol ≠ is used to denote not equivalent.

In the BBC, the functions AND, OR and NEQV operate on the store named and the accumulator. The result is left in the accumulator. The type bits of the two operands do not affect the functions and the type bits of the accumulator remain unchanged.

a	¬a
0	1
1	0

This is the NOT function. It changes noughts to ones and ones to noughts. The symbol ¬ is used to denote NOT.

In the BBC, NOT 24 means replace the C(accumulator) by a content formed from C(24); ones replacing noughts and noughts ones. The type bits of the accumulator are to be the old type bits of 24.

It is difficult to illustrate the usefulness of these instructions in the context of simple examples but the following should illustrate some of the techniques that are available.

When the library routine PRINT is obeyed at a time when the accumulator contains an S-word, a piece of program similar to the following is entered. When reading this, imagine that the accumulator initially contains the characters XY*? so that in binary C(A) would look like this:- | 11 | 011000 | 011001 | 110010 | 110111 | . After each instruction has been obeyed, C(A) is shown again to help you to follow.

129	CYCR-6	11	011001	110010	110111	011000	
130	OPUT 1	11	011001	110010	110111	011000	X is output
131	CYCR-6	11	110010	110111	011000	011001	
132	OPUT 1	11	110010	110111	011000	011001	Y is output
133	CYCR-6	11	110111	011000	011001	110010	
134	OPUT 1	11	110111	011000	011001	110010	* is output
135	CYCR-6	11	011000	011001	110010	110111	
136	OPUT 1	11	011000	011001	110010	110111	? is output

Notice that the accumulator is restored to its original value.

The next example illustrates how READ might have been written. It is a simplification of the actual routine which has to deal specially with the quote and unquote characters. This time, assume that the next four characters of data are AB*? and notice that the C(A) and C(NEXT) are shown in character form rather than in binary.

		Accumulator				NEXT			
54	IPUT 1				A	rubbish			
55	SHFR-6			A		rubbish			
56	IPUT NEXT			A					B
57	OR NEXT			A	B				B
58	SHFR-6		A	B					B
59	IPUT NEXT		A	B					*
60	OR NEXT		A	B	*				*
61	SHFR-6	A	B	*					*
62	IPUT NEXT	A	B	*					?
63	OR NEXT	A	B	*	?				?

The next example accepts as data a passage of English terminated with a full stop. The number of letter 'A's in the passage is counted and printed out.

```
16  TAKE 0
17  PUT COUNT
18  IPUT CHAR
19  TAKE+1    (i.e. the 'value' of the character A)
20  NEQV CHAR
21  JNZ 23    (i.e. Jump if not an 'A')
22  INCR COUNT
23  TAKE+42   (i.e. the 'value' of the full stop character)
24  NEQV CHAR
25  JNZ 18    (i.e. Jump if not a full stop)
26  CAPTN
27  <NO. >
28  <OF '>
29  <A's=>
30  TAKE COUNT
31  PRINT
32  STOP
```

Examples 11

11.1 Write a program which counts the number of times the word 'AN' appears in a passage of English. Use the following as test data:

AN ANTELOPE AND AN ANT RAN DOWN THE STRAND.

11.2 Assume that the following program is run with data +11 and +13. Work through it by hand and say what the output would be. What operation, in general, does the program perform?

```
    16 READ
    17 PUT N1
    18 READ
--> 19 PUT N2
|   20 NEQV N1
|   21 PUT ANS
|   22 TAKE N2
|   23 AND N1
|-  24 JEZ 29
|   25 SHFR-1
|   26 PUT N1
|   27 TAKE ANS
|-  28 JUMP 19
|-> 29 TAKE ANS
    30 PRINT
    31 STOP
```

Acc	N1	N2	ANS
001011	001011	001101	
001101			

(Work with the six least sig. bits)

Chapter 12 P-words and changing Types

Sometimes, it is useful to be able to change the type bits of a cell content. There are two instructions in the order code which enable the user to manipulate types:

CHYP N means 'change the type bits of N. The new type bits are to be the two least significant bits of the accumulator.'

TYPE N means 'copy the type bits of cell N to the least significant end of the accumulator.'

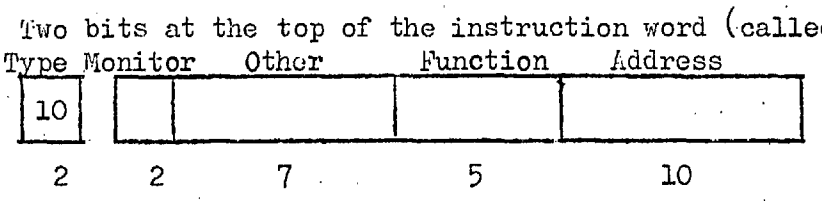
Consider the following program. It reads in 11 items of data and prints them out again the reverse order. Each item is placed in a separate cell, the first item goes into 110, the second into 109 and so on.

```
    16 TAKE 43  }  These four instructions serve no purpose the first
    17 PUT 21   }  time the program is run.  On subsequent occasions,
    18 TAKE 44  }  they restore C(21) and C(31) to their correct values.
    19 PUT 31   }
 ┌─>20 READ
 │  21 PUT 110    This instruction becomes different each time round the
 │  22 TAKE 0                                                      loop.
 │  23 CHYP 21    Change C(21) to an I-word.
 │  24 DECR 21
 │  25 NEQV 45
 │  26 JEZ 30     Jump when C(21) has reached the value PUT 100.
 │  27 TAKE+2
 │  28 CHYP 21    Change C(21) back to a P-word.
 └─ 29 JUMP 20
 ┌─>30 LINE
 │  31 TAKE 100   This instruction becomes different each time round the
 │  32 PRINT                                                       loop.
 │  33 TAKE 0
 │  34 CHYP 31    Change C(31) to an I-word.
 │  35 INCR 31
 │  36 TAKE 31
 │  37 NEQV 46
 │  38 JEZ 42     Jump after C(21) has taken the value   TAKE 110
 │  39 TAKE+2
 │  40 CHYP 31    Change C(31) back to a P-word.
 └─ 41 JUMP 30
 └─>42 STOP
    43 PUT 110  }  Dummy instructions which are never obeyed.
    44 TAKE 100 }  The first two are used to restore corrupted
    45 PUT 100  }  instructions and the last two are used
    46 TAKE 111 }  for comparing.
```

This program is laborious but the technique is important because it illustrates the power of the computer. Remember, a computer is a machine that will process information; in particular, it can process the information (i.e. the program) which is doing the processing. The ideas involved in this example are so important that there are automatic methods available for achieving this action. These will be discussed in chapter 15.

Chapter 13 Monitoring a Program

Programs often fail to run correctly first time and it is sometimes difficult to find out what has gone wrong. Although the computer has been able to output messages to show up programming faults, many logical errors cannot be detected by the computer and the programmer is forced to check through the program by hand. This is often a laborious process when programs are long and complicated. To save the programmer time and trouble at this point, certain facilities have been provided so that the computer can give a running commentary of its progress.



monitor bits) are normally zeros and play no part in the definition of the instruction. Ones placed here however, create at run time an exit to a special monitor routine which has the task of causing the instruction to be obeyed normally as well as displaying extra information to the user. Three different kinds of monitoring are available corresponding to 01, 10 and 11 in the monitor bits. The commands 'TRACE', 'CHECKA' and 'CHECKN' are used to set these bits and the command 'REMOVE' removes them. 'TRACE', 'CHECKA' or 'CHECKN' on a line by itself causes the appropriate monitoring bits to be set up on all jump instructions. Programs with errors often get stuck in a loop so this is an easy method of finding out which loop. In a long program, it is sometimes a good idea to let it get stuck first, interrupt it and set up 'CHECKA' (say), and then allow it to continue. Alternatively, a monitor command can be followed by a space and then a list of addresses separated by commas and terminated by a new line. In this case, monitoring bits are set in the instructions whose addresses appear in the list. The exact format of the additional output is shown in the appendix. Briefly, 'CHECKA' monitors the accumulator and 'CHECKN' the named store whilst 'TRACE' simply indicates the path traced by a program.

Examples 13

13.1 Re-run examples 10.4 and 10.5 with monitoring so that you can see how quickly they converge.

Chapter 14 Extending the order Code

The second accumulator

A second accumulator (with address 2) is provided in the BBC. If a function mnemonic is followed immediately by a 2, when the instruction is being translated a special bit is set in the instruction word. Later, when such an instruction is obeyed, computation takes place in the normal way except that A2 (accumulator 2) is used as one of the operands instead of A1. As it is possible to specify the address of an accumulator in the address field of the instruction, it is possible to pass data between the two accumulators. Having two accumulators available saves unnecessary taking and putting but does not otherwise increase the power of the computer.

X-mode

In a similar way, an X in front of a function mnemonic sets a special bit in the instruction. This time, this has the effect of exchanging the accumulator content and the content of the named store after the function has been obeyed. Thus reading numbers from data and adding them into a store location called SUM can be done as follows:

```
READ                          READ
XADD SUM      instead of      ADD SUM
JUMP                          PUT SUM
                              JUMP
```

X-mode defined in this sense is useless as far as Jump instructions are concerned, but it has been possible, here, to extend the function code by providing a set of skip instructions. A skip instruction uses the named store content and the accumulator to state a condition instead of the accumulator alone. If the condition fails, the next instruction is obeyed as with a jump instruction, but if the condition succeeds, the SCR is increased by an extra one and the next instruction but one is obeyed. In the example which illustrates these techniques, a loop of instructions has been created which uses A2 as a counting cell so that the loop is repeated for C(A2) taking values 100, 101, 102 .... 200. It is usual for the instructions within the loop to use C(A2) but not to change its value.

```
                                    
                                    
                                    
25 TAKE2+99      C(A2):= 99
26 INCR 2        A2 is treated as a normal cell here
27 ----------
28 ----------
                                    
                                    
49 ----------
50 SKAE2+200     Jump to 52 if C(A2) = 200
51 JUMP 26       Closes the loop.
52 ----------
                                    
                                    
```

The instruction in cell 50 means skip if C(A2) = 200 otherwise obey the next instruction. The constant 200 is loaded into a cell from the top end of the store and the address of this cell is assembled into the program.

NEG MOD and CLR

These three functions leave the C(named store) unchanged; in each case the result is left in the accumulator.

NEG	N	means	C(A):= -C(N)
MOD	N	means	C(A):= the positive value of C(N)
CLR	N	means	C(A):= 0

XCLR is a particularly useful instruction. When answers are accumulated in a cell, it is necessary to clear it first, the first 3 instructions in the worked example of chapter 5 could have been replaced by the single instruction XCLR 100.

Examples 14

14.1 Evaluate $ax^3 + bx^2 + cx + d$. x, a, b, c and d are to be read from the data and the answer is to be printed out. Try to solve the complete question so that the stored program is as short as possible. (Aim to do it in about 12 instructions).

14.2 Each line of the given table is worked out from the previous line from the statements

```
x:= k*x + 1.0
y:= k*y
k:= k + 1.0
e:= x/y
```

Write a program that will print out the table continuing until two successive values of e are the same (within the accuracy of the computer).

X	Y	K	E
1.0000	1.0000	1.0000	1.0000
2.0000	1.0000	2.0000	2.0000
5.0000	2.0000	3.0000	2.5000
16.000	etc.		

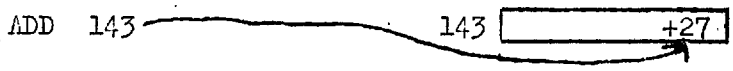
Chapter 15 Arrays

In Chapter 12, we solved a problem by allowing a program to modify itself. Basically, we needed a set of instructions which could be obeyed a number of times but with small differences each time through. The methods used in Chapter 12 necessitated the restoring of instructions to their initial values so that the program could be run successfully a second time. In this chapter, we shall consider automatic methods which allow instructions to take temporary and variable addresses (and to a lesser extent temporary and variable functions). The fact that the changes are only temporary means that we do not need to restore instructions to their initial value and the fact that they are automatic makes them quicker and easier to use. In the BBC there are three methods.

Indirect addressing

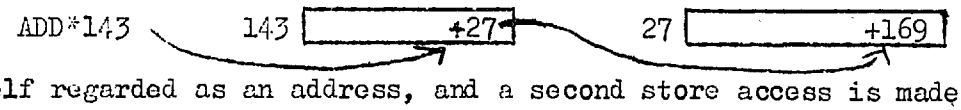
A special bit in the instruction word (called the indirect bit) is set to a one if an asterisk (instead of a space) separates the function from its address. At run time, this bit has a special effect.

When an instruction is written normally,



the operand (in this case +27) is pointed to by the address stored in the instruction. We say, in this case, that the address 143 is used directly.

When an asterisk is used, the content of the cell pointed to, is

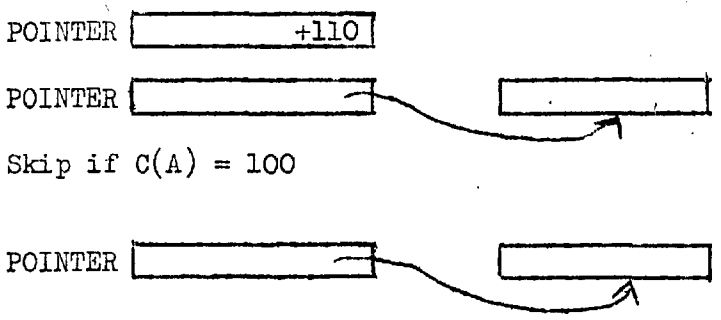


itself regarded as an address, and a second store access is made. In this case

```
      ADD 143      means      add C(143)
                           ie add + 27
but   ADD*143      means      add C(C(143))
                           ie add C(27)
                           ie add + 169.
```

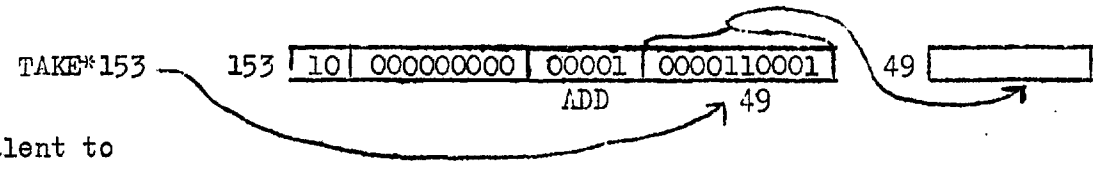
We will now use this technique to rewrite the example from chapter 12.

```
16 TAKE+110
17 PUT POINTER
18 READ
19 PUT*POINTER
20 DECR POINTER
21 SKAE+100
22 JUMP 18
23 INCR POINTER
24 TAKE*POINTER
25 PRINT
26 LINE
27 TAKE POINTER
28 SKAE+110
29 JUMP 23
30 STOP
```



Notice that this solution is much shorter.

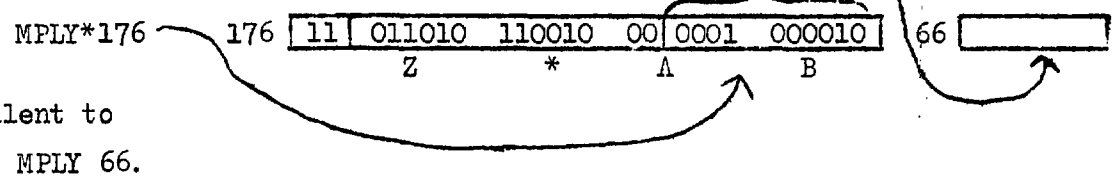
So far, an I-word has been used to specify the indirect address but it is possible to use P-words and S-words as well (but not F-words). As only a 10 bit address is sought, only the 10 least significant bits of the store content are used. If C(153) is the instruction ADD 49,



is equivalent to

TAKE 49

and if C(176) is an S-word specifying the characters Z*AB,



is equivalent to

MPLY 66.

Index modification

(Before going onto this part of the chapter, attempt example 15.1). An alternative to indirect addressing is the modification of an address by an index register.

PUT 29: 14

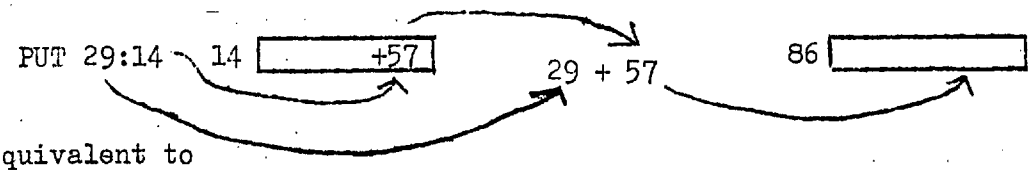
gets translated into

Type	Index	Function	Address
10	1110	10001	0000011101

At run time, the operand (instead of being obtained from cell 29) is obtained from a cell whose address is

29 + C(14)

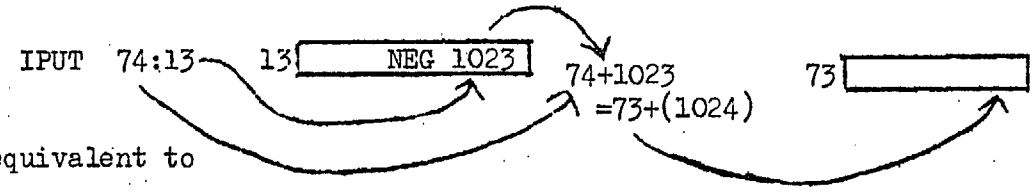
So that, if C(14) = 57 the instruction



is equivalent to

PUT 86

if the content of the index register is a P-word or an S-word, the effect is the same except that the final address is cut down to 10 bits after the addition has taken place.
If C(13) is the instruction NEG 1023



is equivalent to

IPUT 73

It should be noted that only 4 bits of the instruction word have been reserved to specify an index register. Because of this, only cells with addresses from 0 to 15 can be used. Many of these cells have special properties, and their uses as index registers have special effects. In particular, using 0 as an index register implies no modification since C(0) is always 0.

As an example using index registers, the problem from chapter 12 is repeated again.

```
   16 TAKE2+11
→  17 SUBT2+1      A2 takes the values 10, 9, 8 .....0
   18 READ
   19 PUT 100:2    A2 is used as an index register
   20 JNZ2 17
→  21 PRINT
   22 LINE
   23 ADD2+1       A2 takes the values 1, 2, 3 .....11
   24 SKAN2+11
   25 STOP
→  26 TAKE 100:2   A2 is used as an index register.
   27 JUMP 21
```

There are two functions which use the index bits in a special way.

LDN 100:14 means C(100):= C(14)
and LDR 100:14 means C(14):= C(100)

In both these cases the assembler faults an instruction if no index register is specified. Using LDR, we can replace the first instruction in the above example by

16 LDR+11:2

Modifying an indirect address

For the specialist, we shall now explain the meaning of an instruction such as

TAKE*ARRAY:15

Briefly, indirect access takes place before modification, so that in this case, C(C(ARRAY) + C(15)) is used as an operand.

The Execute instruction

If an instruction is required with a variable function, neither of the methods described so far will suffice. Suppose we wish to search the characters from the data until we come to a '+', a '-' an '*' or a '/' and we then wish to ADD, SUBT, MPLY or DVD accordingly.

```
25 IPUT 15
26 TYPE 100:15
27 SUBT+2
28 JNZ 25          Jump back if C(100 + C(15)) is not an instruction.
29 TAKE N1
30 EXEC 100:15     Obey the instr in 100+C(15).  Do not alter the
                     SCR unless a Jump is found.
```

```
100 +0
101 +0
 .   .
 .   .                  This is a table
 .   .
144 +0                  which is used by
145 ADD N2
146 SUBT N2             the rest of the

147 +0                  program.
148 +0
149 +0
150 MPLY N2
151 DVD N2
152 +0
 .
 .
 .
163 +0
```

An interrupt to the error routine will occur at run time if an attempt is made to use an EXEC instruction which does not have a P-word as an operand. An error also occurs if an attempt is made to execute an instruction which itself is an execute instruction.

Examples 15 (use indirect addressing and index modification. Both methods are important.)

15.1 Assume that numerical values have been placed in the cells 101 to 110. Write a routine which will scan this area of store and search for the biggest value. Arrange for the answer to be output in the form

C(107) = 29.8

(this answer assumes that 29.8 was the largest value and that this item was found in cell 107.)

15.2 By considering what happens at bit level, verify that TAKE 100:15 when C(15) = -7 is equivalent to TAKE 93.

15.3 Write a program which will input numbers and output them in ascending order of size.

15.4 Adapt 15.3 so that it will rearrange the following in alphabetic order.

⟨NOW⟩⟨IS⟩⟨THE⟩⟨TIME⟩⟨FOR⟩⟨ALL⟩⟨GOOD⟩⟨MEN⟩⟨TO⟩⟨COME⟩⟨TO⟩⟨THE⟩⟨AID⟩⟨OF⟩⟨THE⟩⟨AA⟩

15.5 Write a program to evaluate standard deviation.

15.6 Write a program which uses the EXEC and TYPE instructions with items from the given table. The program is to count the number of vowels, number of consonants and number of words in a sentence. The full stop is to be the terminating character.

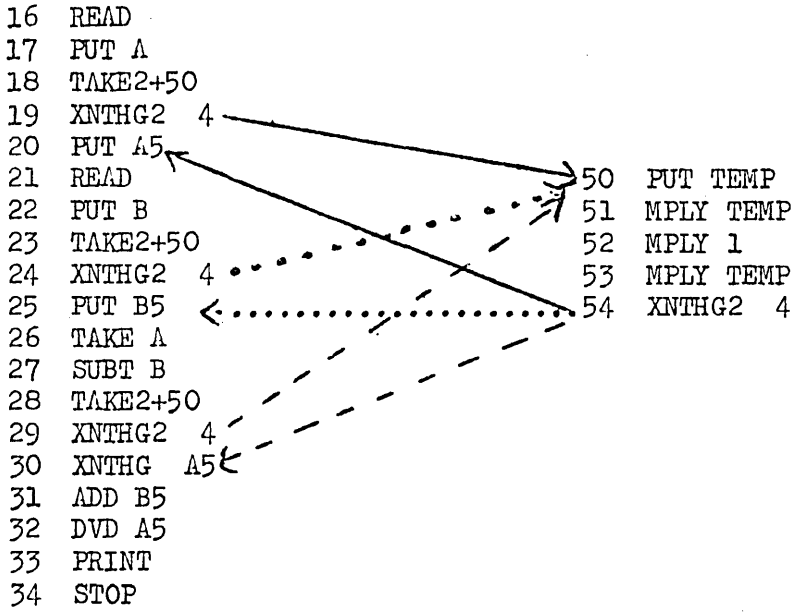
```
101 INCR VOWEL   A
102 INCR CONS    B
103 INCR CONS    C
104 INCR CONS    D
105 INCR VOWEL   E
 .               .
 .               .
 .               .
126 INCR CONS    Z
142 JUMP 60      Full stop.  Substitute an appropriate exit address.
161 INCR WORD    space
162 INCR WORD    new line
```

Chapter 16 Subroutines and Relative Addressing

Suppose we want to write a program to evaluate

$$\frac{a^5 + b^5}{(a-b)^5}.$$

In the program, the task of raising to the fifth power has to be done three times, and it is useful if the work can be organised so that the same set of instructions can be obeyed from different places. It is possible to do this within the framework of what has been learnt so far. Consider the following solution, remembering that XNTHG means swap.



In this solution, A2 is used to store the address to which we must return after instructions 50 to 54 have been obeyed. The following table shows the content of SCR and the content of the A2 after the current instruction has been accessed, after the SCR has been incremented and before and after the current instruction has been obeyed.

Address of current instr.	Current instr.	Before		After		Address of next instr.
		C(SCR)	C(A2)	C(SCR)	C(A2)	
19	XNTHG2 4	20	50	50	20	50
54	XNTHG2 4	55	20	20	55	20
24	XNTHG2 4	25	50	50	25	50
54	XNTHG2 4	55	25	25	55	25
29	XNTHG2 4	30	50	50	30	50
54	XNTHG2 4	55	30	30	55	30

The instructions in cells 50 to 54 are called a closed subroutine, and the technique demonstrated in this example is so important that special instructions are provided for storing the return address.

JLIK 50

means jump to the instruction in 50 and remember where you have come from (i.e. store the return or 'link' address). The return address is stored in cell 7 (cell 7 has this special property) so that exit from the sub-routine is made by using the instruction

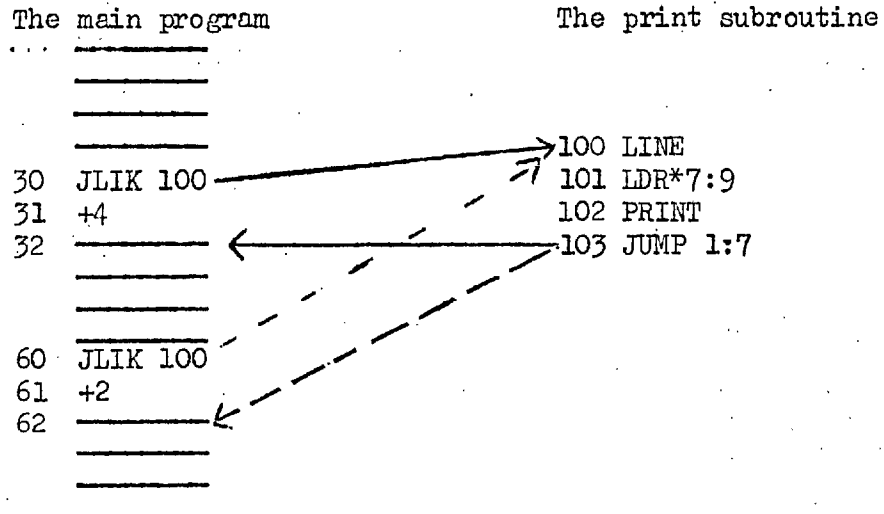
JUMP*7

i.e. jump to the instruction whose address is stored in 7.
The previous example is now shown again using JLIK.

```
16  READ
17  PUT A
18  JLIK 50
19  PUT 2   C(A2):= a^5
20  READ
21  PUT B
22  JLIK 50
23  XADD 2  C(A2):= a^5+b^5
24  TAKE A
25  SUBT B
26  JLIK 50
27  DVD2 1  C(A2):=  C(A2)/C(A1)
28  PRINT2
29  STOP

50  PUT TEMP
51  MPLY TEMP
52  MPLY 1
53  MPLY TEMP
54  JUMP*7
```

If more than one piece of information needs passing to a subroutine we can use A2. Alternatively, the following technique is useful. A subroutine is required which will print the accumulator content on a new line to a variable number of places. Numbers are normally output to 5 places, but this number can be varied by altering the content of cell 9. C(9) is used by PRINT to specify the number of places that numbers are to be printed to. It has a presumed value of +5 and this value is set by the LOGIN routine.



In this program, the first number is printed to 4 places and the second number to 2.

Relative addresses

When instructions are being assembled into store, the assembler normally checks to make sure that they are going in sequence. An occasional break in this sequence is achieved by using the command 'BREAK'. 'BREAK' also specifies a new base for relative addressing. The address of the first instruction to follow 'BREAK' is regarded as a base and any addresses given from here onwards may be written relative to this base. Relative addresses are terminated with a + sign and have the base address added to them as they are translated. The base address itself is stored in cell 12. If the first address after 'BREAK' is itself a relative address, the new base is set to be the next address in sequence. Because of this, the only relative address which can immediately follow 'BREAK' without error is 0+ and this means no break in sequence at all. 'EDIT' allows

instructions without regard to sequence and after 'EDIT' the computer keeps no record of the address of the last instruction stored. It follows that the sequence

```
'EDIT'
_____
_____
_____
'BREAK'
0+
```

is undefined and will result in an error.

If subroutines are written with all but the address of their first instructions in relative form they can easily be relocated so that they can be reassembled into different parts of the store.

Examples 16

16.1 Incorporate the print subroutine given in this chapter into the example 14.2 so that x is printed to 8 places, y to 7, k to 2 and e to 5. Specify x, y and k to be I-words, but use the library routine FLOAT to convert them to floating point form just before the DVD instruction.

16.2 Write a program to evaluate the area of a triangle given the co-ordinates of the three vertices. Write a subroutine to evaluate the lengths of the sides using the formula $\sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}$.
The addresses of the x and y values (not the values themselves) are to be placed in the four cells following the JLIK instruction. The length of the side should be in A1 on exit. The area of the triangle can be found, once the sides are known, by using the formula

$$\Delta = \sqrt{S(S-a)(S-b)(S-c)}\ , \qquad S = \tfrac{1}{2}(a+b+c).$$

16.3 What special precautions need to be taken when one subroutine calls another.

16.4 Write a subroutine which when obeyed reads 6 numbers from the data and stores them in consecutive cells. The address of the first cell is to be in A1 when the subroutine is entered.

Write a second subroutine which prints 6 numbers. Again, the address of the first is in A1 on entry, but the address of each subsequent number is found by adding 6 to the previous address.

Use these two subroutines to transpose a 6 x 6 matrix.

Chapter 17 'ASSIGN' and 'MESSAGE'

The assembly code that we have been using so far, like all assembly codes, becomes tedious if large programs are to be coded. To save programmers the trouble of coding at this level, most modern computers are equipped with compilers. A compiler is simply a program which is able to translate generalised statements straight into the machine code of a computer. Instead of the programmer having to write the details of the program, the compiler does it for him. Most compilers hide the machine code version of a program from the user, who sees only the high level statements of his own. The BBC, however, is not equipped with a full compiler so the user is shown, at assembly level, the code that the compiler commands generate.

'ASSIGN'

This command is followed by an arithmetic assignment statement and is used to evaluate algebraic formulae. If identifiers A, B and ANS have been declared, an assignment statement might look something like this:

```
'ASSIGN'  ANS:= (A+B)*(A-2*B)
```

It would generate code which would assign to C(ANS) the value of [C(A) + C(B)]*[C(A) - 2*C(B)]. It is not possible any longer to use lower case letters or C( ) for 'content of'; the assignment statement as given above is in accordance with modern conventions.

The code corresponding to this statement is compiled on to the end of the user's existing program and this code is immediately typed back to him. In this case, if his existing program had finished with an instruction in cell 50 the following might have been typed:

```
51  JOI   52
52  TAKE  1022  (A)
53  ADD   1020  (B)
54  PUT   1016  (BBC1)
55  TAKE  1015  (  +2)
56  MPLY  1020  (B)
57  PUT   1012  (BBC2)
58  TAKE  1022  (A)
59  SUBT  1012  (BBC2)
60  PUT   1012  (BBC2)
61  TAKE  1016  (BBC1)
62  MPLY  1012  (BBC2)
63  PUT   1018  (ANS)
```

Clear the overflow indicator

Work space declared automatically

BBC2:= 2*B

BBC2:= A-2*B

Although assignment statements look so much like ordinary algebra and although the normal order of precedence is observed, there are a number of restrictions and pitfalls. These are listed below:

1. The type of the operands may affect the result. In particular, integer division may occur unless care is taken to avoid this.

2. Assignment statements must be short enough to fit onto one line.

3. Because everything must be typed at one level, ↑ is used to denote 'to the power of' and / to denote division.

4. Multiplication and division have equal precedence so that A/B*C means (A/B)*C and not A/(B*C).

5. The multiplication sign must always be present. AB can only represent a single identifier and not the product of A and B.

6. Two operators are not allowed together. A*-B must be written as A*(-B).

7. Superfluous brackets do not constitute an error, but the number of closed brackets must match the number of open brackets.

8. Any of the library routines may be used in an assignment statement but those marked with an asterisk in table 3 of the appendix make no sense. Each library mnemonic must be followed by an operand in brackets. The PRINT routine is particularly useful. A:= PRINT(B) + SQRT(A+B) means print the value of B and assign the value of B + $\sqrt{A+B}$ to A in place of its current value.

'MESSAGE'

This is best illustrated by an example. If the last instruction to be stored had been placed in cell 70, 'MESSAGE' <MARY HAD A LITTLE LAMB> would cause the following to be compiled and output:

```
71  LIBR 15   (CAPN)
72  <MARY>
73  < HAD >
74  < A L >
75  <ITTL >
76  <E LA >
77  <MB>
```

Since there is no 'next instruction' after 'EDIT', neither 'MESSAGE' nor 'ASSIGN' can be used after 'EDIT' without an intervening 'BREAK'.

Examples 17

17.1 Write a program to solve a quadratic equation if a, b and c are to be provided as data. Make sure that your answer copes with the case a=0 as well as with $b^2<4ac$. Your answers should be appropriately headed.

17.2 Write a program to compute the nth Fibonacci number (see Ex. 7.1) using the formula

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

17.3 An approximate value for n! is given by the formula

$$\text{fact}(n) \doteq \sqrt{2\pi n}\cdot\left(\frac{n}{e}\right)^n \qquad \text{(where } e \doteq 2{\cdot}7183\text{)}$$

Write a program to evaluate this formula and the percentage error from the true value. n is to be read in as data.

17.4 Write a program to find the value of

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!}$$ for values

of x read in as data. The expression should be nested. [Hint: nesting means that an expression such as $ax^3 + bx^2 + cx + d$ is rewritten as $((ax + b)x + c)x + d$].

Chapter 18 Loops

'LOOPN'

It is usual for high level languages to provide automatic facilities for allowing a piece of program to be repeated a number of times. Within such a loop a particular variable often takes different values each time through. Typically, the loop must be entered at the beginning (so that initial values can be set up). From here, it either works its way through to a defined end or it is abandoned part way by some jump to another separate piece of the program.

In the command language, 'LOOPN' is able to set up the necessary code to establish a loop. This code is added to the rest of the user's program and is typed back to him just as it is for 'ASSIGN'. 'LOOPN' is followed by a counting variable and three arithmetic expressions which represent respectively the initial value of the counting variable, the step size or amount the counting variable must be increased by each time round and the number of times the loop is to be repeated. Thus if a user typed in

'LOOPN' ANS:= A+1:SQRT(B+C)/B:X-1

code would be compiled which, when obeyed, would cause a set of instructions to be obeyed with ANS taking an initial value of A+1 and subsequent values increased successively by $\sqrt{\frac{B+C}{B}}$. The whole process would be set up to be obeyed X-1 times.

When the code corresponding to 'LOOPN' had been output to the user, he would be invited to type in the instructions corresponding to the loop itself. This might be done by typing individual instructions or by using other automatic programming facilities but eventually, the stage would be reached when it was necessary to terminate the loop. To do this, the user would need to type the command 'REPEAT' followed by the name of the counting variable. This results in the compilation of a jump instruction to close the loop, and the insertion of a forward jump reference into the part already compiled.

The user is not constrained to write loops with a constant step size; in our example $\sqrt{\frac{B+C}{B}}$ could be varied by altering one or both of the variables B and C within the loop. In the same way, X might vary but this does not matter since the number of times through is compared each time with a freshly worked out value of INT(X-1).*

'LOOPV'

Frequently, a user requires much less than the generous facilities just described. Under these circumstances, 'LOOPN' will produce very inefficient programming. Clearly, it is wasteful to evaluate X-1 each time round the loop if once would have done. Under such circumstances, he would be well advised to use 'LOOPV' rather than 'LOOPN'. He would use this in the same way but the code generated would be more efficient. He would not now have the ability to vary the step size within the loop.

* INT is a library routine which finds 'the whole number part of'.

It is important that the distinction between 'LOOPV' and 'LOOPN' is clearly understood. Compiler designers have difficulty in deciding whether to provide flexible or efficient facilities. In the example given below, we are required to sum

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots\ldots \text{ to n terms}$$

where x and n are to be read off the data. 'LOOPN' must be used because x↑r/factr varies each time round the loop. We say that the step size and n are 'called by name'.

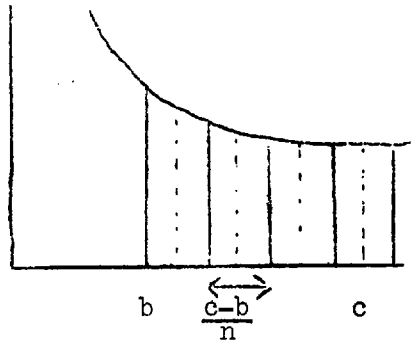
Only the source code (i.e. the part the user types) is shown here.

```
'LOGIN' SMITH
'DECLARE' X, N, R, FACTR, SUM
16 XCLR R
17 TAKE+1
18 PUT FACTR
19 READ
20 PUT X
21 READ
22 PUT N
'MESSAGE' <SUM TO N TERMS=>
'LOOPN' SUM:=1.0:X↑R/FACTR:N
'BREAK'
0+INCR R
1+TAKE R
2+XMPLY FACTR
'REPEAT' SUM
'BREAK'
0+TAKE SUM
1+PRINT
2+STOP
```

In this next example we are to find an approximation to the area under the graph of

$$y = \frac{1}{\sqrt{x^2-a^2}}$$

where x goes between b and c. Data consists of a, b and c and the answer is to be compared with the theoretical answer which is $\log_e\left(\frac{c+\sqrt{c^2-a^2}}{b+\sqrt{b^2-a^2}}\right)$. The method we are going to use involves dividing the area into strips, finding the total length of all the strips and multiplying by the strip width. Clearly, the accuracy of the method will depend on the number of strips, this we shall call n and will also be read in from the data.



In the diagram, n is shown as 4 and the total strip length is the sum of the dotted y-values.

The program for this question is now given. Notice that this time, 'LOOPV' is used because the step size and n do not vary. We say that they are 'called by value'. If we had used 'LOOPN' instead, we should still get the right answer but it would take longer.

Again, only the source code of the program is given.

```
'LOGIN' BLOGGS
'DECLARE' A,B,C,X,YSUM,AREA
16 READ
17 PUT A
18 READ
19 PUT B
20 READ
21 PUT C
22 READ
23 PUT N
24 XCLR YSUM
'LOOPV' X:=B+(C-B)/(2*N):(C-B)/N:N
'ASSIGN' YSUM:=YSUM+1.0/SQRT(X*X-A*A)
'REPEAT' X
'MESSAGE' <AREA BY SUMMATION=>
'ASSIGN' AREA:=PRINT(YSUM*(C-B)/N)
'BREAK'
0+ LINE
'MESSAGE'<AREA BY THE FORMULA=>
'ASSIGN' AREA:=PRINT(LN((C+SQRT(C*C-A*A))/(B+SQRT(B*B-A*A))))
'BREAK'
0+ STOP
```

Examples 18

18.1 Write a program to tabulate n, $\sqrt{n}$, n^2, $n!$ for values of n going from 0 to 10.

18.2 By nesting one loop inside another, write a program to tabulate the volumes of cylinders of varying height and radius. Arrange for your answers to be output in a rectangular array with suitable headings.

Chapter 19 Features for the Specialist

It is not expected that many readers will concern themselves with all the details explained in this chapter. They are put here mainly for the sake of completeness and for the benefit of those who wish to look closely at some of the concepts raised elsewhere.

Integer Overflow

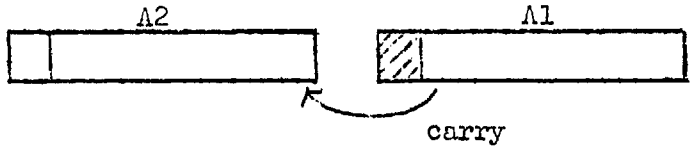
In chapter 9, we considered what happened when an attempt was made to store integers that were too great in magnitude to fit into one cell. Normally, this constitutes an error, the instruction is cancelled and an exit to the error routine automatically takes place. In certain cases, however, integer overflow is legal and deliberate. The case described in chapter 6, where a sum-check was made, is typical of one such class of problems. In the BBC, when integer overflow occurs, cell 5 is inspected.

If this cell contains zero, the program is faulted, but if it is greater than zero, the program continues. Each time overflow occurs c(5) is increased by 1 and this content can be inspected by the program so that the current state of overflow is known.

A special instruction is provided for testing for overflow. JOI is a jump instruction which causes a jump only if c(5)≠0. JOI also has the effect of clearing c(5).

Double length working

Multiplication is a particular problem as far as overflow is concerned. In general, two n-bit integers yield a product 2n-bits long. **Our computer** , in common with many real computers, makes special provision for this. Double length working replaces, automatically, the normal single length + overflow multiplication instruction if c(5) is set to a negative value. This is achieved by regarding the two accumulators as a single 48 bit cell. Two single length operands are provided in the normal way via A1 and



the named store. The double length answer is stored so that the least significant part is in the 23 least significant bits of A1 and the most significant part is in A2. The sign bits of both accumulators are correctly set and double length overflow (unlikely) is faulted.

Integer division is similar. This time, the dividend is double length and the divisor and quotient are single length. There is a difficulty here, however, because the quotient might well overflow the single length accumulator. Should this happen the program is faulted and an overflow message is displayed. No remainder is kept after double length division and the result is not rounded.

It is not appropriate to specify A2 explicitly when double length working is used.

Integer Division and the remainder

When integer division (single length) takes place, the remainder is left in cell 3. This presents no difficulty when dividend and divisor are both positive, but in other cases two valid answers are possible.

$\frac{16}{-3}$ = -5 remainder +1 . or -6 remainder -2

but $-\frac{16}{3}$ = -5 remainder -1 or -6 remainder +2

One surprising consequence of this is the fact that

$\frac{a}{-b}$ is no longer equal to $\frac{-a}{b}$.

The BBC always gives as answer to single length integer division a result with a positive remainder.

INT and FRAC

The library routines INT and FRAC adopt the same philosophy over signs as that taken by integer division.

INT(-17.34) = -18 but INT(17.34) = 17

and FRAC(-17.34) = +0.66 but FRAC(17.34) = 0.34

In all cases INT(A) + FRAC(A) $\fallingdotseq$ A.

Difficulties can arise over rounding errors and any F-word is increased by a small tolerance evaluated from c(6) before it is truncated.

Undefined Instructions

There are certain logical inconsistencies relating to such instructions as

INCR 1

The definition in table 1 states that C(A) remains unchanged but c(named store) is increased by 1. Both of these are not possible when the accumulator and the named store are both the same cell. One way out of such difficulties is to specify the other accumulator so that in this case

INCR2 1

is well defined. However, the following instructions are particularly useful and are given the stated meaning by the computer.

NEG 1	a:=-a
MOD 1	a:=|a|
NOT 1	a:=¬a
OPUT 1	output LSE of a
IPUT 1	a:= next character
INCR 1	a:=a+1
DECR 1	a:=a-1

NEG2 etc. have similar interpretations.

Chapter 20 The multiprogramming computer

There are certain problems associated with the EBC that arise because an on-line user is sharing the computer with others. It is clearly not a good idea if the computer is being shared to make available to the ordinary student the full capabilities of the machine. Whilst it may be desirable to give the user as much freedom as possible, including perhaps the freedom to corrupt or destroy his own program, it would create havoc if we allowed him to interfere with other users' programs.

In the EBC, users are protected from one another by dividing the store into self-contained blocks of 1024 cells and only allowing the user to address cells within his own block. A moment's thought, however, will show that other pieces of program within the machine must be given the power to break these barriers. The assembler, for instance, receives instructions in mnemonic form on behalf of a particular user and places

their binary equivalents in the user's store, not in its own and the user himself is able to get out of his own program and into the assembler by using the interrupt facility.

This chapter is concerned with some of the difficulties inherent in a multiprogramming environment and solutions are offered in terms of the BBC configuration. For reasons of efficiency, most of the suggestions made have not been implemented, but the associated problems remain. The Elliott computer itself is used in many places rather than the simulated computer for although the real computer is less satisfactory than the hypothetical computer in many ways, it has the undoubted advantage that it is some 200 times as fast.

BBC M - The multiprogramming computer

The store consists of a number of blocks numbered from 0 upwards, each of 1024 cells. All of these blocks are basically the same as the computer the student has been programming but block 0 has a number of cells with special hardware features which are used to organise the transfer of control (jump) from one block to another.

Each block contains its own accumulators, sequence control register and other index registers and the content of these cells is left undisturbed when an exit from a block is made. At any one time, only one block is active for the computer has only one central processor. The active block is normally the one from which the current instruction has been taken.

The whole store content must now be thought of as one big program. Some parts of this program are primarily there to process other pieces of program rather than numeric data. Some parts of the overall program are relatively permanent, other parts 'belong' to users and vary. So that the computer is able to keep control of all these pieces, block 0 contains a very special piece of program that master minds the rest. This special piece is called the supervisor. Thousands of man-years have been spent writing supervisors and some of these are very sophisticated and represent the ultimate in the present state of the art of programming.

The problems with which we must now deal fall into three categories:

(a) Jumping from one block to another.
(b) Choosing an operand from a block other than the currently active block.
(c) Input and output.

(a) Transfer of control between blocks

This occurs when the following conditions arise:

(i) Time has run out. User programs cannot be allowed to run to completion because they might be very long or get stuck in a loop. Block 0 must contain a clock cell which can count instructions. Every time an instruction is obeyed the c(clock cell) is decreased by 1. When it goes negative, an automatic jump is made to a special point within the supervisor which must reset the clock and sort out which block is to be serviced next.

(ii) A Library routine is obeyed. These are entered via the supervisor on behalf of a user or on behalf of the assembler or a command. Library routines are always run to completion, the exit from the block calling the routine is the only action taken by the machine code instruction LIBR, the rest is built-in software. It might have been more accurate to describe LIBR as a block jump instruction. The STOP instruction is a special Library routine because it causes a more permanent exit to a block, it is an instruction to the supervisor that this block is to receive no more time slices until further notice. It is important to notice that STOP is not taken literally by the computer. Only a hardware failure would cause the computer to stop completely.

(iii) A user has interrupted either his own program or some other piece of program which is working for him.

(iv) A run-time 'error' has occurred. In small computers, certain errors are so horrific that the computer stops, possibly with some indicator lit on the control panel. Other errors can be dealt with by program. In BBC M all errors cause automatic entry to the supervisor which in turn jumps to an error routine which is responsible for the output of any appropriate message.

(v) A program has run out of data.

(vi) A monitored instruction has been attempted.

The setting up of monitor bits in an instruction effectively make it illegal. Entry to the error routine is via the supervisor and it is from here that the monitoring routines are entered. The instruction to be monitored is extracted and interpreted, results are put back and the extra print-out is arranged.

(b) The operand cell

The absolute address of any cell within the computer is made up from its block number and its 10 bit address. When accumulator 1 is being used, the block no. for any operand is understood to be the same as the block number of the current instruction. The same is not necessarily so for instructions involving A2. The supervisor can control a special cell called the operand cell. This cell is normally made to contain the block number of the user being serviced so that access to cells can be made across the block boundaries.

(c) Input and output

The difference in speed between the user and the teletype on the one hand and the computer on the other presents difficulties. To avoid the computer from being kept waiting, buffer areas able to take a whole line of input and a number of characters of output are set aside for each user. In this way a user can type in a line of information at leisure during which time the computer is virtually ignoring him. It is only when this line is complete that the user gets a time slice. In the same way, a user has a number of characters for output generated for him by some program segment but the output of these characters only starts when their assembly is complete. During their output, that user gets no time slice. The library routine READ is able to take advantage of the buffering because it can access this buffer area by address and not just via IPUT. READ is able, in particular, to go back and read a character twice.

There are other difficulties associated with BBC M which have not been mentioned. What happens if an error state arises within the error routine itself? What happens when information is being transferred to or from film and a number of others **questions. For those who have** managed to get this far through the manual it is time to advise them to study a real computer.

General Questions

1. Discuss the merits and demerits of automatic programming compared with low-level programming.

2. Write a set of subroutines for manipulating complex numbers. Subroutines should be written to read, print, add, subtract, multiply and divide complex numbers. Pairs of adjacent cells are to be used to store the numbers and these pairs should be specified by the lower address; it is this cell that should be used to store the real part of the number. Where appropriate, the addresses of the two operand pairs should be in the accumulators on entry. On exit A1 should store the real part of the result and A2 the imaginary part.

Write a suitable program to test the routines.

3. A better method for finding the area under a graph than that described in chapter 18 is to use Simpsons Rule. The area, which is divided into an even number of strips, is approximately equal to

$$\frac{h}{3}\left[\text{first} + \text{last} + 4*\text{even} + 2*\text{odd}\right]$$

where h is the strip width and first, last, even and odd refer to the ordinates of the strip boundaries.

4. Consider the following segment of program. It is a very elementary supervisor which is capable of running concurrently two programs simultaneously within the user's store. One program is to start at location 300 and one at 600. The following restrictions apply to these two slave programs.

(i) No subroutine entries can be made using JLIK unless the link address is rescued from location 7. This is because the supervisor uses location 7.

(ii) One program will be unable to read data and neither should print results. Answers should be left in ANS1 and ANS2 and will be printed out at the end.

(iii) STOP must not be used. JLIK 36 should be used instead.

(iv) Both programs should contain the instruction JLIK 37 at regular intervals (say every ten instructions). This subroutine organises the transfer from one slave program to the other.

(v) Programs should not use common work space or work space used by the supervisor. The only exceptions to this rule are the two accumulators and the SCR.

Write two programs which satisfy those restrictions apart from (iii) and (iv). Make sure they work. Adapt the programs to satisfy (iii) and (iv) and enter the supervisor. The programs should now run together.

```
16  TAKE+300
17  PUT SCR
18  TAKE+21
19  PUT N
20  JUMP 600
21  XNTHG A1
22  JUMP 45
23  JUMP* 7
24  JUMP 25
25  CAPTN
26  <ANS1>
27  <=>
28  TAKE ANS1
29  PRINT
30  CAPTN
31  <ANS2>
32  <=>
33  TAKE ANS2
34  PRINT
35  STOP
36  INOR N
37  EXEC*N
38  XNTHG A2
39  PUT TEMP
40  TAKE SCR
41  XNTHG 7
42  PUT SCR
43  TAKE TEMP
44  JUMP*7
45  INCR N
46  XNTHG A1
47  JUMP 38
```

Sets up initial values.

Dummy instructions obeyed by EXEC*N.

Entered when both programs have finished.

When both programs are running, this routine does the swapping.

Used when first program finishes.

APPENDIX

Key to terms used

N	the address stored in the address part of an instruction
L	the address of the cell which stores an instruction
C( )	the content of the cell whose address is given
n	C(N) before an instruction has been obeyed
n'	C(N) after an instruction has been obeyed
A	The accumulator (this may be followed by a subscript if a particular accumulator is implied)
a	C(A) before an instruction has been obeyed
a'	C(A) after an instruction has been obeyed
R	an index register
r	C(R)

Command Word Summary

(A command word is always braced with apostrophes and can occur

(a) at the beginning of a session
(b) after another command word
(c) during a Read instruction
(d) in place of an instruction)

The following command words are used to control the system:

1. 'LOGIN' Initiates a new user giving him a 'clean sheet'. 'LOGIN' is followed by the user's name (only recognised names are accepted). The logging in sequence must be followed by either another command word or the beginning of the program.

2. 'BREAK' Initiates a new segment of program. Allows a break in address sequence. Addresses terminated by a plus sign are relative to the first instruction after the last 'BREAK'.

3. 'RUN' Causes the current program to run from the first instruction given. 'RUN' N causes the program to run from the instruction in location N. Before running the program, the overflow indicator is cleared.

4. 'CONTINUE' Causes the computer to continue from the point where it was interrupted.

5. 'EDIT' After 'EDIT', the computer is ready to receive instructions without regard to sequence.

6. 'TELEP' Input is now via the teleprinter. All tapes prepared off-line should be terminated with 'TELEP'.

7. 'PTAPE' Causes a message to be displayed at the computer centre so that the operator can load a new tape that has been prepared off-line.

8. 'FILE' n Overwrites the existing C(file) with a copy of the program in Core store. n must take one of the values 1, 2 or 3 corresponding to one of the 3 slices of backing store available to each user.

9. 'FETCH' As 'FILE' except that the Core store is overwritten by C(file). Certain useful programs are available to all users on a read only basis. These may be brought off backing store by using 'FETCH' followed by the name of the program. Any user who has such a program is asked to submit it (fully documented) for inclusion.

10. 'ADVANCE' + n, N, M. Moves forward the C(store) between N and M by n places. The operands of P-words pointing to words that have been moved are automatically adjusted (provided that they are not referred to indirectly or by an address which is modified by an index register). n must be a signed integer.

The following command words are used to provide diagnostic help:

11. 'TRACE' followed by an address list. Sets up TRACE bits in the locations named. 'TRACE' on a line by itself sets up TRACE bits on all JUMP and SKIP instructions. When the TRACE bits of a P-word are set, each time the instruction is obeyed, its location address is output followed by an asterisk.

12. 'CHECKA' Similar to 'TRACE'. CHECKA bits present in a P-word which is obeyed cause L and a' to be output preceded by a new line and an asterisk.

13. 'CHECKN' Similar to 'CHECKA' except that L and n' are output preceded by two asterisks. A <u>single</u> identifier can follow 'CHECKN' in place of an address list. This causes the appropriate monitoring bits to be set up on all instructions calling the identifier directly.

14. 'REMOVE' N, M. Removes monitoring bits from instructions in the range N to M. 'REMOVE' on a line by itself removes all monitoring.

15. 'LIST' Causes an annotated copy of the current program to be output on the teleprinter together with the necessary command words for the program to be successfully re-input. 'LIST' N, M outputs the segment of program from N to M.

16. 'DUMP' N, M. Outputs the non-zero C(store) from N to M.

The following command words provide high level programming facilities:

17. 'DECLARE' Followed by a list of identifiers causes the computer to reserve locations at the top end of the store for variables with the given names. Identifiers can also be declared by default when a warning message is output to guard against mis-spelling.

18. 'PLACES' + n Causes C(9):=n so that answers are printed out to n places. Emergency printing is output if this is not possible.

19. 'ASSIGN' ROOT:=(-B + SQRT(B↑2 - 4*A*C))/(2*A). Sets up BBC code to assign the appropriate value to ROOT. This machine code is added to the end of the user's program and is output to complete the user's version. Identifiers may not be declared by default in an assignment statement. Single length Multiplication and division are used during integer arithmetic.

20. 'LOOPN' A:=B:C:D. Inserts a piece of BBC code to set up a loop so that a routine may be repeated with A taking the values B, B + C, B + 2*C .... B + (D-1)*C i.e. D times altogether. A, B, C and D may be any declared identifiers. B, C and D may be replaced by constants or by arithmetic expressions.

21. 'LOOPV' Similar to 'LOOPN'. Code compiled under the control of 'LOOPV' is more efficient but C and D are evaluated once and for all and may not vary inside the loop.

22. 'REPEAT' A Closes the loop which uses A as a counting variable.

23. 'MESSAGE' <THIS IS A MESSAGE> Compiles one or more caption routines together with the appropriate S-words. The message cannot be more than one line long. C(11) is unaffected.

The Hypothetical Computer

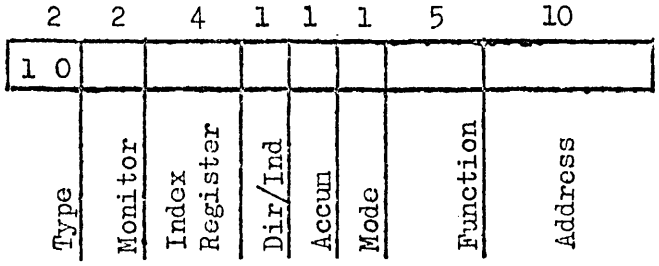
The following is the description of the computer as the user sees it. It is not a description of the Elliott 803 but of the hypothetical computer which has been simulated on the Elliott. This hypothetical machine has been designed from an Educational standpoint but although no such real machine exists, there is no reason why a machine should not be made to this design.

The cell length is 26 bits. Two of these play no part in the arithmetic but describe the way in which the other 24 bits are to be interpreted.

The four different types are given below:

1 0	A P-word.	For storing a line of program or an instruction.
1 1	An S-word.	For storing a string or list of characters.
0 0	An I-word.	For storing an integer.
0 1	An F-word.	For storing a floating point number.

Instruction (P-word)



The user has addresses from 0 to 1023 available for his exclusive use. In these he must store his program and leave room for his data and work space. The ways in which the components of the instruction word affect the arithmetic are given below:

Monitor The presence of 1's in this part of the word causes extra output to be given during the execution of the instruction. The purpose of the monitoring facilities is to provide the user with some method of detecting logical errors in his program at run times. Three different types of output are possible:

0 0		No monitoring
0 1	TRACE	L is output each time the instruction is obeyed followed by an *
1 0	CHECKA	L and a' are output preceded by a new line and an *
1 1	CHECKN	L and n' are output preceded by a new line and two asterisks

Monitoring can be switched on or off at any time by using the appropriate command word.

Index Register Any of the first 16 addresses can be used as an index register. The effect of an index register is to cause the function to use C(N + r) as operand instead of C(N), or if the address is used indirectly C(n + r) instead of C(n). Neither N nor C(L) are affected by the presence of the index register. As C(Zero) = zero (always) the effect of zero in the index part of the instruction is to cause no address modification.

Direct/Indirect addressing If this bit is a '0' the function operates on C(N), but if it is a '1' a second store access is made using the 10 least significant bits of C(N) as address. The C(N) must not be an F-word.

Accumulator Two accumulators are available. 'A1' is used if 0 appears in the instruction word and 'A2' is used if a 1 appears.

Mode If this bit is a '0' the function operates under direct or D-mode which is described in the function table. If it is a '1', X-mode operates which causes a' and n' to be exchanged after the arithmetic has been performed. On transfer control functions, the effect of X-mode is to change a jump instruction into a skip instruction, which means that (provided some condition is satisfied) the Sequence Control Register is incremented by two instead of the usual one.

Function The following table describes the operation of the full order code under D-mode and the last eight functions under X-mode.

For each of the first 16 functions, n':=n, for the next eight, C(A) and C(N) are both affected, the next seven alter the value of the SCR and the last function, because it is concerned with loading and reading from an index register, must be followed by an indexed address.

Function Table (Order Code)

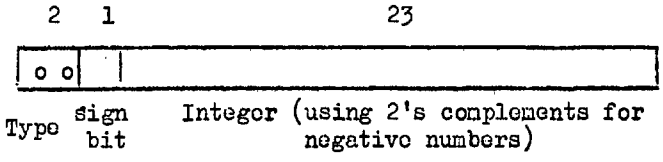
TABLE 1

Function	Mnemonic	D-mode result	P-word permitted	I-word permitted	F-word permitted	S-word permitted	Mixed Arith result	Remark
0	NTHG*	a':=a n':=n	✓	✓	✓	✓		
1	ADD*	a':=a+n n':=n	x	✓	✓	x	F-word	
2	SUBT*	a':=a-n n':=n	x	✓	✓	x		
3	MPLY*	a':=a*n n':=n	x	✓	✓	x		Double length working for integer x and . available but not normal
4	DVD*	a':=a/n n':=n	x	✓	✓	x		
5	TAKE*	a':=n n':=n	✓	✓	✓	✓		
6	NEG*	a':=-n n':=n	x	✓	✓	x		
7	MOD*	a':=|n| n':=n	x	✓	✓	x		
8	CLR*	a':=0 n':=n	✓	✓	✓	✓		a' an I-word
9	AND*	a':=a∧n n':=n	✓	✓	✓	✓	Type bits are not affected	
10	OR*	a':=a∨n n':=n	✓	✓	✓	✓		
11	NEQV*	a':=a≠n n':=n	✓	✓	✓	✓		
12	NOT*	a':= ¬n n':=n	✓	✓	✓	✓		
13	SHFR*	right shift n times on c(accumulator)	n must be an I-word					-24 ≤ n ≤ 24
14	CYCR*	cyclic shift n times on c(accumulator)	a can be any type					
15	OPUT*	a':=a n':=n	x	✓	x	✓		character corresponding to 6 least significant bits of n is output
16	IPUT	a':=a n':=next character				✓		
17	PUT	a':=a n':=a	✓	✓	✓	✓		
18	INCR	a':=a n':=n+1	x	✓	✓	x		
19	DECR	a':=n n':=n-1	x	✓	✓	x		
20	TYPE	type bits of n to LSE	✓	✓	✓	✓		
21	CHYP	type bits of n' are the two LS bits of a	✓	✓	✓	✓		
22	EXEC	Obey the instr. in N	✓	x	x	x		n must not be an EXEC instr.
23	LIBR	Affects C(A) only	Address part used to specify which subroutine				No X-mode available	

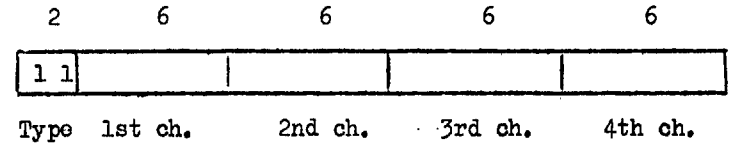
	Direct Mode		X-Mode	
	Mnemonic	Effect	Mnemonic	Effect
24	JLIK	C(7):=L+1 Jump	SLIK	n':=L+1 Skip(ie obey instruction in L+2 next)
25	JUMP	Instruction in N obeyed next	SKET*	Skip if a and n are same type
26	JEZ	Jump if a = o	SKAE*	Skip if a = n
27	JNZ	Jump if a ≠ o	SKAN*	Skip if a ≠ n
28	JLZ	Jump if a < o	SKAL*	Skip if a < n
29	JGZ	Jump if a > o	SKAG*	Skip if a > n
30	JOI	Jump if overflow indicator is set: Clear indicator	SNLZ	Skip if n < o
31	LDN	n':=r a':=a r':=r	LDR*	r':=n a':=a n':=n

* These functions may be followed by a constant (of an appropriate type) instead of by the usual address.

Integer (I-word)



String (S-word)



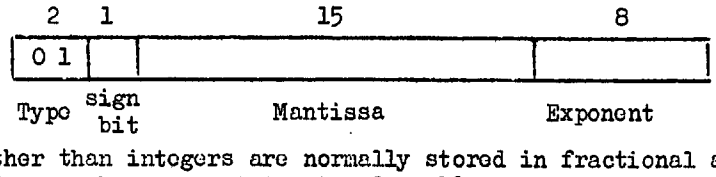
Up to 4 characters can be stored in packed form in a given location. The character representation is given in the table below and is independent of any code that might be used for tape preparation.

TABLE 2

Decimal	Octal	Binary	Character	Decimal	Octal	Binary	Character	Decimal	Octal	Binary	Character	Decimal	Octal	Binary	Character
0	00	000 000	*Bell	16	20	010 000	P	32	40	100 000	0	48	60	110 000	un-quote
1	01	000 001	A	17	21	010 001	Q	33	41	100 001	1	49	61	110 001	apost-rophe
2	02	000 010	B	18	22	010 010	R	34	42	100 010	2	50	62	110 010	*
3	03	000 011	C	19	23	010 011	S	35	43	100 011	3	51	63	110 011	/
4	04	000 100	D	20	24	010 100	T	36	44	100 100	4	52	64	110 100	:
5	05	000 101	E	21	25	010 101	U	37	45	100 101	5	53	65	110 101	)
6	06	000 110	F	22	26	010 110	V	38	46	100 110	6	54	66	110 110	=
7	07	000 111	G	23	27	010 111	W	39	47	100 111	7	55	67	110 111	?
8	10	001 000	H	24	30	011 000	X	40	50	101 000	8	56	70	111 000	↑
9	11	001 001	I	25	31	011 001	Y	41	51	101 001	9	57	71	111 001	←
10	12	001 010	J	26	32	011 010	Z	42	52	101 010	.	58	72	111 010	≠
11	13	001 011	K	27	33	011 011		43	53	101 011	(	59	73	111 011	;
12	14	001 100	L	28	34	011 100	not	44	54	101 100	¤	60	74	111 100	,
13	15	001 101	M	29	35	011 101		45	55	101 101	+	61	75	111 101	space
14	16	001 110	N	30	36	011 110	used	46	56	101 110	-	62	76	111 110	new line
15	17	001 111	O	31	37	011 111		47	57	101 111	quote	63	77	111 111	rub out

* used to represent 'no character'

Floating point (F-word)



Numbers other than integers are normally stored in fractional and exponent form. The exponent is stored as 128 + true exponent. The mantissa is stored, using 2's complements for negative numbers, with the binary point immediately after the sign bit. Zero is always stored as an I-word.

Library Subroutines

TABLE 3

Number	Mnemonic	Operand Types Allowed	Result Type	Remarks
1	SQRT	F or I	F	$a \geq 0$
2	LN	F or I	F	Natural logs. $a > 0$
3	EXP	F or I	F	$a < 128 \log_e 2$
4	READ	-	any	See below for details
5	PRINT	Any	-	
6	SIN	F or I	F	Operand in radians
7	COS	F or I	F	
8	TAN	F or I	F	
9	ARCTAN	F or I	F	Principal value
10	*STOP	-	-	Outputs END OF PROGRAM and returns control to user.
11	*LINE	-	-	Outputs a new line.
12	INT	F (or I)	I	Finds the integer part.
13	FRAC	F	F	Finds the fractional part.
14	FLOAT	I	F	Changes the integer C(A) to floating point form.
15	*CAPTN	-	-	Up to 11 S-words can follow LIBR 15. Those are output to form a caption when Libr 15 is obeyed. The instruction following the last S-word is obeyed next.

* No operand required

READ During a READ instruction:

(a) If quote appears, the quote marker * (cell 11 in the user's store) is set to 1 and the next character is read.

(b) If unquote appears, the quote marker is set to zero and the next ch. read.

(c) If the quotes marker is set to 0 an F or I-word is expected (or a word in the form of 8 octal digits preceded by an I F P or S and enclosed in brackets). A decimal point or subscript 10 causes a number read to be stored as an F-word, otherwise it is stored as an I-word. Numbers can be terminated by any character except a decimal point, digit or subscript 10. The terminating character is the next to be read after the READ instruction has been obeyed. Commas, new lines or spaces are ignored if no number has yet been read.

(d) If the quotes marker is set to 1, characters are read and packed into the accumulator (starting at the most sig. end) as an S-word and input stops when either 4 characters have been read or unquote is read. (Quote and unquote cannot themselves be stored in a string during a READ instruction.

* The quotes marker can be set to 1 or 0 by normal programming.

PRINT During a PRINT instruction, the output given depends on the type bits of a:

P-word. The mnemonic form of the instruction is output. Spaces follow to make up the total number of characters to 15.

S-word. Up to 4 characters are output.

I-word. The integer is output to C(9) places. It is preceded by a space or - sign and terminated with 2 spaces. Leading zeros are replaced by spaces so that a total of C(9)+3 characters is output. If the number is too large for this type of output, emergency printing takes place; the number is printed to 8 places preceded by a new line.

F-word. The number is output to C(9) places. It is preceded by a space or - sign and terminated with 1 space. A decimal point is included so that a total of C(9)+3 characters is output. If the number is too large or too small for this type of output, emergency printing takes place; the number is printed in decimal and exponent form with a 5 figure mantissa and preceded by a new line.

Index registers with special properties

TABLE 4

Address	Initial Setting	Special property ('Hardware')
0	0	Always contains the integer 0.
1	0	Accumulator 1 (designated by a 0 in the acc. bit of a P-word).
2	0	Accumulator 2 (designated by a 1 in the acc. bit of a P-word).
3	0	Remainder register. Contains a + ve remainder after single length division.
4	0	Sequence control register. Contains, as an I-word, the address of the current instruction and is incremented immediately after store access.
5	0	Overflow indicator. An I-word which is incremented each time integer overflow occurs. The overflow indicator is also used to determine the nature of integer MPLY and DVD. If C(5)< 0, double length arithmetic takes place using both accumulators otherwise single length arithmetic takes place.
6	+16	Tolerance setting. (For an F-word, zero is taken to mean that the exponent part ≤C(6))
7	0	Link address stored here after JLIK.

TABLE 5

Address	Initial Setting	Special properties ('Software')
8	+1023	Points to the last address available to the programmer. The part of the store between C(8) and the top end of the store contains constants and cells reserved for declared identifiers.
9	+5	'PLACES' setting for print instruction.
10	0	Stores the address of the first instruction.
11	0	Quotes marker (used during READ).
12	0	Base address for relative addressing. Set by the assembler, every time 'BREAK' is read, by the address of the following instruction.
13-15	0	Available for normal use by the programmer.

The Assembly Code

P-word

A single instruction in its typed form consists of 5 parts:

(i) The address of the cell where the instruction is to be stored. It must either be terminated by a space (when it is regarded as absolute) or by a + sign (when it is regarded as relative to the address of the first instruction or the first instruction given after 'BREAK').

(ii) The function in Mnemonic form. This may be preceded by an X to indicate X-mode and may be terminated by a 2 to indicate that A2 is to be used.

(iii) A space or asterisk. A space is used for direct addressing and an astorisk for indirect addressing.

(iv) The address of the function operand. (This can be absolute or relative.)

(v) An index register. The presence of an index register is optional. If the function operand is terminated by a space, no index register is expected, but if it is terminated by a colon, an index register (from 1 to 15) is obligatory.

A comment can follow an instruction and is ignored by the computer.

Various programming aids are provided. In particular, a + -, open bracket or quote can replace the space or asterisk described above at (iii). The number, S-word or octal constant following is then used as operand instead of the contents of an address. When this facility is used, an instruction such as

24 TAKE+17.6

might be compiled into

24 TAKE 1023

1023 + 17.6

It can only be used when C(N) is unchanged by the function; this is indicated by an * in the order code.

Users are also able to coin identifiers to name cells used for data so that

16 ADD*ARRAY:6

might be compiled into

16 ADD*1022:6.

The word ARRAY itself is stored as an S-word in an adjacent cell. Only the first 3 and last characters are stored. Identifiers beginning with the letters BBC are reserved and may not be declared.

Library subroutines can be used by typing a simple mnemonic so that READ would be compiled into LIBR 4.

S-word An S-word can be stored directly by following the address of the cell where it is to be stored by up to 4 characters in quotes. Characters so stored are left justified.

I-word An I-word can be stored directly by following the address of the cell where it is to be stored by an integer of up to 8 digits preceded by a + or - sign.

F-word An F-word can be stored directly by following the address of the cell where it is to be stored by a number which must contain either a decimal point or a subscript 10.

Octal input An alternative form of input is provided using octal digits. All eight octal digits must be present, they must be preceded by an I, F, P or S to specify type and the whole must be enclosed in brackets.

System Responses

Question mark - The user is invited to type an instruction. He may type a command instead.

Apostrophe - The user must type a command next.

DATA? - Up to one line of data can be typed. If an apostrophe is not during a READ instruction, the command word read routine is triggered.

END OF PROGRAM - Printed out when STOP is obeyed. A command word is expected next.

UNUSED DATA:- - When unused data remains after an END OF PROGRAM message has been displayed or after an interrupt, up to the next 6 characters of data are output.

INTERRUPTED - Displayed after an interrupt.

TRANSFER COMPLETE - Displayed after a successful transfer to or from film.

FILM BUSY - Displayed if another user is searching on film.

SUM CHECK FAILURE - Due to hardware failure, a program has failed to transfer successfully. Also displayed if a non-existant program is retrieved.

FILM FAILURE - The film handlers are incorrectly set up.

Equals sign - Output after 'TELEP' and 'PTAPE'.

Error Summary

1. Too many char. to a line BBC operates on a line by line basis. Only when a line of typing is completed, does the computer start to 'obey' what has been typed. The computer reserves 67 locations to store up to 67 characters corresponding to one line of program, data or command word sequence. If more than 67 characters are typed, error no. 1 is displayed and the whole line is ignored.

2. Wrong char. during Read The way in which data is interpreted is explained in the section on library routines but error no. 2 is displayed if an illegal character is found during a Read instruction. Some of the more common errors are given below.

   (i) Letter etc. when an F- or I-word expected.
   (ii) Two decimal points in a number.
   (iii) Subscript 10 is followed by more than two digits.

   The read routine is also used at various times during translation.

3. EXEC (EXEC) is illegal It is illegal to execute an instruction which itself is an execute instruction. In particular, an instruction may not execute itself.

4. No addr. before function Even when instructions are being loaded into consecutive locations, the address of the location is given first, then the function and finally the function operand.

5. Addr. out of range Error no. 5 is displayed at translation time if any attempt is made to use an address greater than 1023. At run time, addresses referred to indirectly or modified by an index register are marked down to the 10 least significant so that such addresses are bound to be in the range 0 to 1023. Instructions that are being input are never placed in the index registers (0-15). Error no. 5 is displayed if this is attempted either directly or because automatic programming has been attempted

   (i) after 'EDIT' has been used, or
   (ii) before any other instructions have been assembled.

6. Function Mis-spelt Only the spelling given in the order code is allowed, in particular multiply is abbreviated to MPLY and divide to DVD. Simple Mnemonics are, however, available for the various Libr routines (so that SQRT is equivalent to LIBR 1).

7. Missing Function operand Apart from the library routines which operate on the contents of an accumulator, all functions have an address to specify an operand. Sometimes it is possible to replace this address by a constant (the operand itself) but in all other cases the address must be present.

8. Command Mis-spelt This error sometimes arises when an apostrophe is not followed by a command.

9. No ind. after LDR or LDN LDR and LDN cannot be used without an index register.

10. No ind. after Colon When a function operand is followed immediately by a colon an index register is expected. Any other character following a function operand is regarded as part of an ignorable comment.

11. Ind. too Large An index register must be in the range 1 to 15.

12. Identifier not declared All identifiers should be declared before they are used and it is essential that this is done when any high level programming facilities are used. If a user forgets to declare an identifier during low-level programming, it is assumed to be declared by default but error no. 17 is still displayed to guard against mis-spelling.

13. Impermissible constant When any of the first 16 functions are used in Direct mode, LDR is used or a skip instruction is used, the content of the store named as operand remains unchanged. When this is the case, it is sometimes convenient to replace the address part of an instruction by a constant (either an I-word, an S-word, an F-word or a word given in octal). This can be done by replacing the space or asterisk which follows the function by an open bracket, quote, + or - sign followed by the required constant. When this is done, the appropriate constant is stored at the top end of the store and its address is placed in the address part of the instruction. Error no. 13 is displayed if an attempt is made to use this facility with an inappropriate function.

14. No comma after Addr. Sometimes a command word has to be followed by a list of addresses or identifiers. The items in this list must be separated by commas and terminated by a new line.

15. Illegal parameter list When a pair of addresses is used with any of the command words LIST, DUMP, ADVANCE or REMOVE, the 1st address gives the start of the process defined by the command word and the 2nd the end of the process. The 1st address must, therefore, always be less than or equal to the second. A 'REPEAT' parameter must correspond to the counting variable of the most recent unclosed loop.

16. Illegal Addr. Displayed if characters other than digits are found when an address is read, or if an address greater than 1023 is read.

17. No instr. in this cell Unless a jump or skip instruction is obeyed, instructions are obeyed in sequence. Error no. 17 is displayed if the next cell does not contain an instruction (i.e. a P-word). It is important to notice that this is a run time error; it is possible (and sometimes desirable) to translate a sequence of instructions which do not finish with a Jump or Stop instruction.

18. F-word for store access When indirect addressing is used, the store accessed is the one pointed to by the 10 least sig. bits of the cell stated in the address part of the instruction. This cell is normally an I-word but could be an S-word or another P-word. It is an error, however, to use an F-word in this way.

19. Operand of wrong type Certain functions only make sense when they are applied to operands of a particular type. The types available to the various functions are shown in the order code. If error no. 19 is displayed, it should be remembered that it may be the content of the accumulator which is of the wrong type or the content of store.

20. Wrong addr. sequence Instructions are normally stored in sequence, but this sequence can be broken temporarily by using the command word 'BREAK' or permanently using 'EDIT'. Failure to use 'BREAK' when the sequence is broken does not necessarily constitute an error and error no. 20 is only given as a warning. After any error has been displayed, the computer is prepared to accept the next instruction out of sequence so that the user can correct a mistake immediately or leave it till later. Error no. 20 should not be confused with error no. 17; error no. 20 is only displayed at translation time.

21. Floating point overflow numbers stored in floating point mode must satisfy the inequality $-2^{128} \leqslant n < 2^{128}$; any answer (apart from one resulting from division by zero) which is outside this range will cause the computer to stop (without performing the arithmetic) with error no. 21 displayed.

22. Division by zero Displayed if integer or floating point division is attempted by zero.

23. Operand out of range SHFR and CYCR (details given in the order code) demand operands within a certain range.

24. Libr. addr. part too big When the library function is specified in an instruction, the address part of the instruction is used to determine whichever library routine is required. Only 15 routines are available and error no. 24 is displayed if the address part of the instruction word is greater than 15.

25. Places cannot be gr 8 The maximum integer capable of being represented using a 24 bit word (including sign bit) can be represented by an 8 digit decimal number. For this reason, a request for more than 8 places is not allowed.

26. User name not found The system uses the user name to identify
(a) the user's error statistics and
(b) the user's files.
Only recognised users are able to log in and they must use only their recognised name.

27. Illegal Identifier An identifier consists of a number of letters and numerals used as a name to identify a particular cell. A user may invent identifiers at will provided that each begins with a letter and certain system identifiers are avoided.

28. XXXX declared XXXX is replaced by the first 3 and last characters of an identifier that has already been declared. This error could result from an attempt to declare identifiers which were the same in terms of the first three and last characters, (e.g. STRING and STRIKING).

29. 1, 2 or 3 for File no. A user may only specify one of 3 files during a FILE or FETCH command, except that 'FETCH' can be followed by the name of a system program.

30. BBC is reserved Identifiers which begin with the letters BBC are reserved for use during assignment and loop commands.

31. Space overflow Normal programming usually starts from the lower end of the store, work space and constants are allocated cells at the top end of the store. Error no. 31 is displayed if automatic allocation of space at the top end of the store overwrites a P-word or if an attempt is made to store an instruction in a cell already reserved as work space or for a constant. Cell 8 is used to store the address of the last cell that can be used for normal programming.

32. Wrong number of colons The correct number of colons in an assignment statement is 0 and for a loop statement is two. (Apart from the colon in a ':=' sign.)

33. Assignment not complete

34. Mistake at:- Numerous mistakes can be made during an assignment statement, many of the errors cause error no. 34 to be displayed. The six next characters are also output to pinpoint the mistake. A second error will not be detected.

35. Missing bracket Brackets must pair in an assignment statement.

36. Sqroot of neg. no. a must be > or = 0.

37. Log or neg. or zero no. Only logs of positive numbers can be found.

38. Exp operand too large a > 128 $\log_e 2$ would cause floating point overflow.

39. No quotes round message

40. Integer overflow An attempt has been made to perform integer arithmetic leading to a result which is too large (or too small) to be represented in a BBC cell. If the overflow is to be ignored, type 'CONTINUE', but remember, future cases of overflow will not stop the machine.

I N D E X