APPLICATION PROGRAMMING SYSTEMS SO FRIENDLY THAT EVEN A SALESMAN CAN USE THEM

Professor W. M. McKeeman

Rapporteurs : Mr. J. Aspden
Dr. J. Eve
Mr. D. McGlade

Professor McKeeman remarked that the development of the microcomputer appeared to be following the same path as the macrocomputer and making the same mistakes but three times faster. Some of these errors lie in the way microcomputers are designed, and should be comparatively easy to avoid. However, we should ask whether there are any new problems which did not occur before. The answer appears to be that there are, and Professor McKeeman illustrated this by referring to a manufacturer, who was not a sophisticated computer user but one used to machines of great mechanical and electromechanical complexity, but who discovered that much of the machinery which they were experienced and equipped to build could be replaced by a microcomputer. The device in question is a 'sidewalk bank teller'. Their prototype bank teller was delivered with 28K of hand-coded assembly language in it, and was so successful that the bank who received it asked for two more, but also requested an extra feature be incorporated. The manufacturers were already using total memory of the machines but re-organised the program and managed to add this feature.

This went on for four months, with every machine a little different, until the manufacturer realised it was programming for customers and would need an exponentially increasing number of programmers. What could be done to solve this problem? The manufacturer was prepared to dedicate one salesman to each sale so the solution proposed was to turn the salesman into "programmers". A second problem to be overcome was the willingness of the salesman to sell anything, irrespective of the feasibility of producing it. The engineering department was receiving orders, signed by the managerial staff, for items that simply could not be made in any reasonable period of time.

Figures 1 and 2 illustrate the proposed solution. The engineering department produced a program, the "specification processor" which was available to salesmen, via a portable terminal. This program posed a set of questions which the customer and salesman answered; from these answers a complete specification was produced in a Polish-string form which could then be passed to a "product simulator" program. Using the latter the customer and salesmen could check that the specification did indeed meet the customer's needs. If unsatisfactory aspects of the specification were discovered, these could be corrected by repeating these two steps.

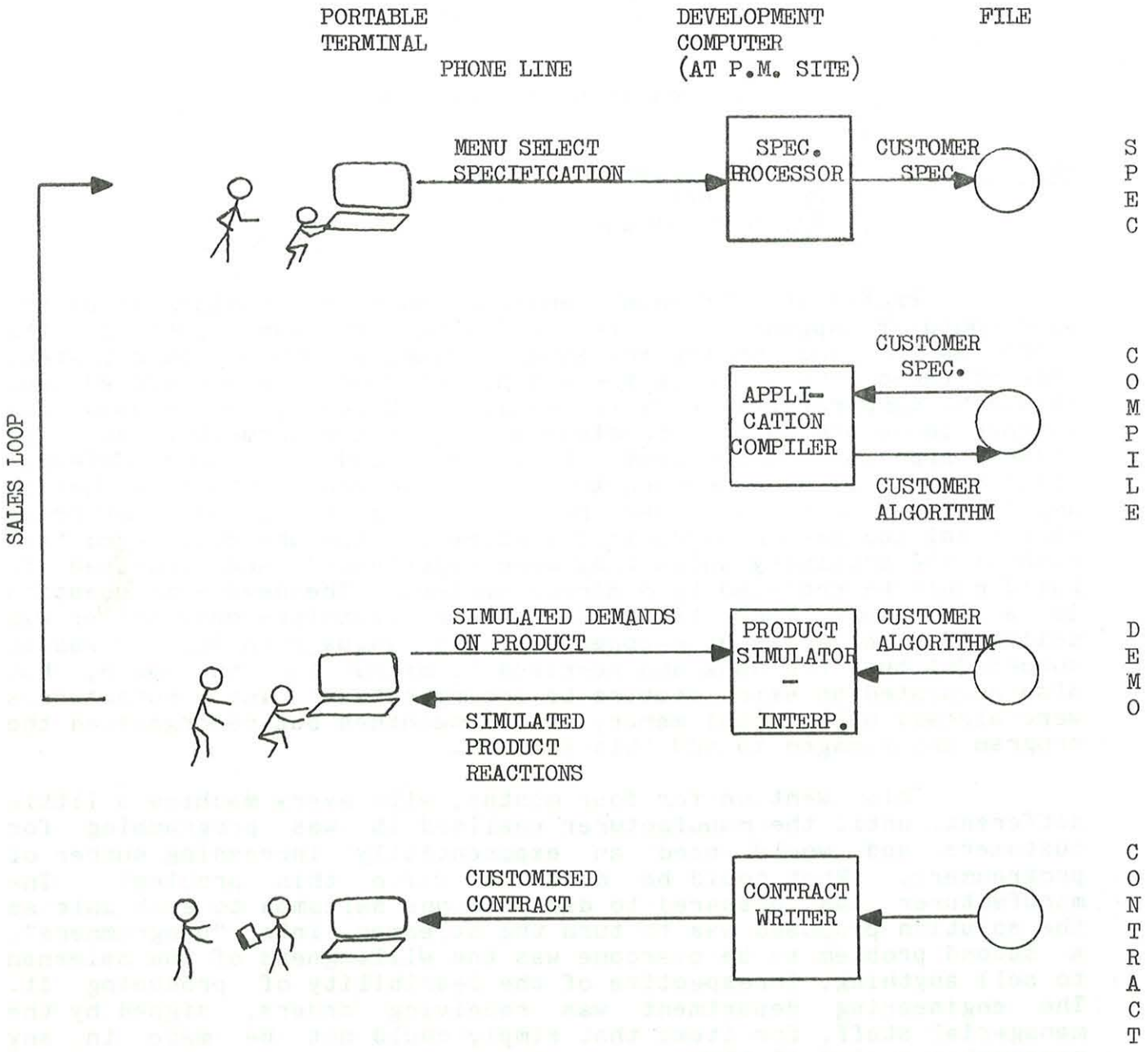

PORTABLE TERMINAL
PHONE LINE
DEVELOPMENT COMPUTER (AT P.M. SITE)
FILE
MENU SELECT SPECIFICATION
SPEC. PROCESSOR
CUSTOMER SPEC.
SPEC
CUSTOMER SPEC.
APPLI-CATION COMPILER
CUSTOMER ALGORITHM
COMPILE
SALES LOOP
SIMULATED DEMANDS ON PRODUCT
PRODUCT SIMULATOR - INTERP.
CUSTOMER ALGORITHM
SIMULATED PRODUCT REACTIONS
DEMO
CUSTOMISED CONTRACT
CONTRACT WRITER
CONTRACT

Figure 1

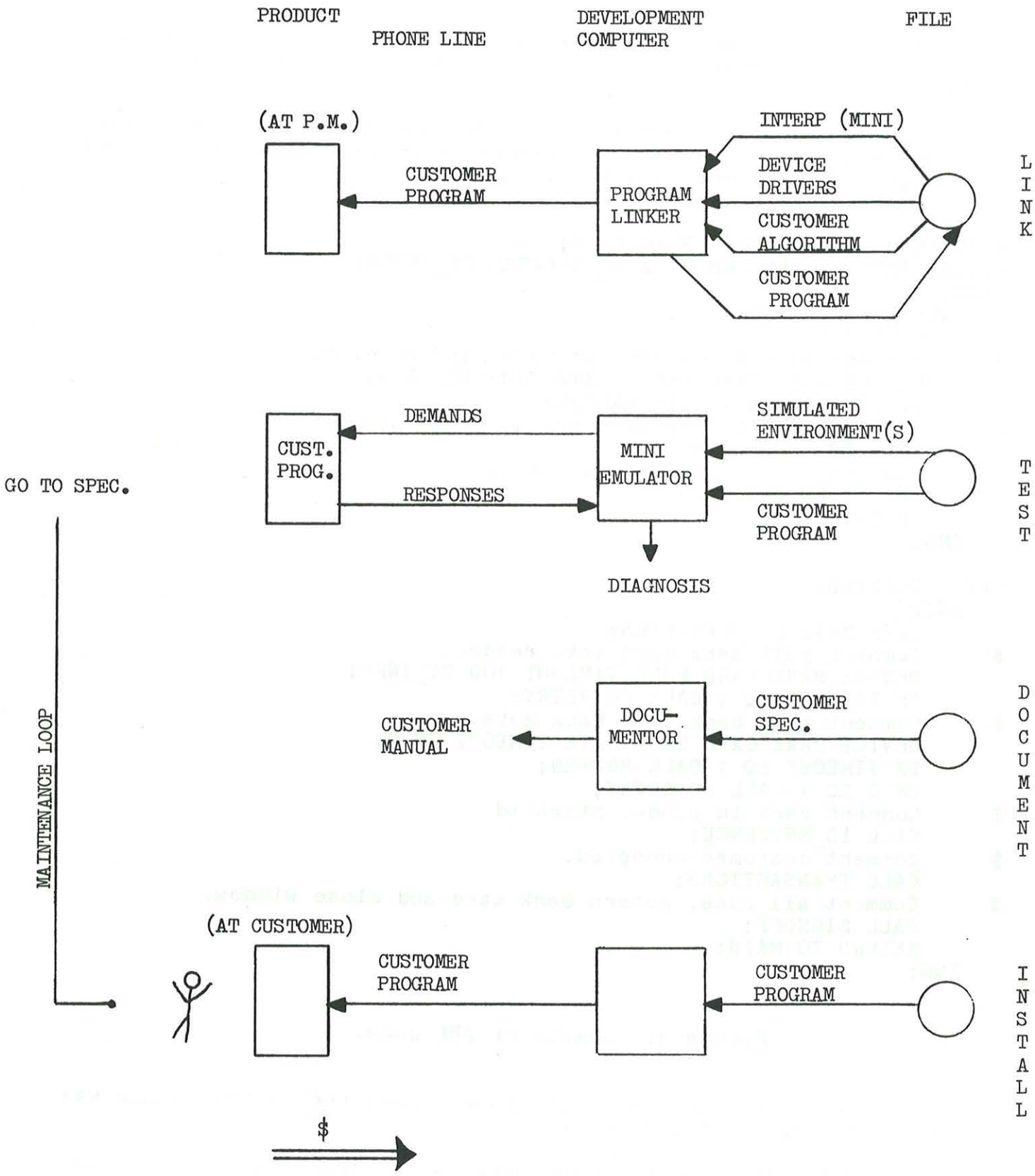

PRODUCT
PHONE LINE
DEVELOPMENT COMPUTER
FILE
(AT P.M.)
INTERP (MINI)
CUSTOMER PROGRAM
PROGRAM LINKER
DEVICE DRIVERS
CUSTOMER ALGORITHM
CUSTOMER PROGRAM
LINK
DEMANDS
RESPONSES
CUST. PROG.
MINI EMULATOR
SIMULATED ENVIRONMENT(S)
CUSTOMER PROGRAM
TEST
DIAGNOSIS
GO TO SPEC.
MAINTENANCE LOOP
CUSTOMER MANUAL
DOCU-MENTOR
CUSTOMER SPEC.
DOCUMENT
(AT CUSTOMER)
CUSTOMER PROGRAM
CUSTOMER PROGRAM
INSTALL
$

Figure 2

On achieving a satisfactory specification the Polish string specification was passed to another program, the "contract writer" which produced an English version of the specification in contract form.

This set of programs was a new tool for the salesman which had the great merit that since engineering produced it, he could not sell anything they could not make.

```
DECLARE B 1 CC_INFO 10 RETRIES 1;
ACCESS RETRIES ONLY FROM CARD_INSERTED CC_RETRY;
MAIN:
   ENTRY;
      CALL RESET;
$     comment test front gate of bank card receiver.
      DEVICE BANK CARD FRONT GATE TIMEOUT 10 B;
      IF TIMEOUT EQ 1 CALL BROKEN;
$     Comment if there is no card, do diagnostics.
      IF B EQ 1 CALL IDLE;
$     Comment start customer id cycle.
      CALL CARD_INSERTED;
      REPEAT MAIN;
   END;

CARD_INSERTED:
   ENTRY;
      DATA MOVE 0 TO RETRIES;
$     Comment pull bank card into reader.
      DEVICE BANK CARD PULL TIMEOUT 100 CC_INFO;
      IF TIMEOUT EQ 1 CALL CC_RETRY;
$     Comment test bank card back gate.
      DEVICE BANK CARD BACK GATE TIMEOUT 10 B;
      IF TIMEOUT EQ 1 CALL BROKEN;
      IF B EQ 1 CALL CC_RETRY;
$     Comment card in place, start id
      CALL ID_SEQUENCE;
$     comment customer accepted.
      CALL TRANSACTIONS;
$     Comment all done, return bank card and close window.
      CALL SIGNOFF;
      RETURN TO MAIN;
   END;
```

Figure 3: Sample of ABL code.

The customer at this stage signed the contract, and was not involved again until delivery.

The next step in developing a program to run on the minicomputer in the bank teller for a specific customer was the construction of a package consisting of a standard interpreter and the user specification together with any device drives that are required. For convenience the package is exercised on a large machine over a wide range of environments. The same specification, once debugged, is given to a documentor program to produce both

maintenance and user manuals for the customer. The maintenance manual in particular includes the modifications needed to change messages, this being a frequent requirement. Finally, this package is sent to the minicomputer at the user site.

Apart from these programs (the specification processor, product simulator, contract writer, standard interpreter and device driver modules, documentor and environment simulator) which are rarely altered, there are no programmers involved in the production cycle. These major programs are not trivial and the specification of environmental tests was particularly difficult. Figure 3 illustrated the language invented to assist the testing. Most of this involved the device driver interactions as approximately half of the commands are device commands.

In summarising Professor McKeeman noted that the main points to observe in the problem and its solution are:

1) The machine selected by the bank teller manufactuer was inadequate for the task. In particular, it was too small and not designed for interpreter use.
2) The solution had a high initial cost but would automatically generate the exponentially increasing amount of code needed.
3) The salesmen were disciplined to sell only feasible products.
4) Maintenance was greatly simplifed. In particularly it enabled the minicomputer at a site to be changed easily, thus localising the fault to hardware or software.

Professor Randell asked if this method could be used in other applications. Professor McKeeman said that for problems like banking it is sufficiently general. Although the menu selection program was large it was easier to develop than to change the product. However there are a lot of problems with too much variability for such an approach. There are cases in which the cost of getting started with this method exceeds that of a more conventional approach.

Professor Pyle wondered if any software company had expressed an interest in the method as a speculative venture. Professor McKeeman replied that if a company wanted such a suite of programs then it would be worthwhile, but there was no obvious way to generalise the method. The menu-selection in particular appears to be very ad hoc. The rest of the system can be built fairly readily.

Professor Pyle pointed out that there was an opportunity to capture, at the point where the salesman and customer are simulating demands, what the customer anticipates the end-users are likely to do.

Professor McKeeman admitted this had escaped him and it would have assisted the construction of the simulated environments which had proved most difficult. The actual writing of the programs was not too involved and had been tried as a student project.

Professor Wells asked if the menu selection had much in

common with computer assisted instruction. Professor McKeeman agreed that there was much in common but did not see how this could be exploited.

Dr. Glaser said most of the modules described have occurred in CAT systems over the last fifteen years but such systems had eventually found the menu selection approach to be a limiting factor.

Professor McKeeman agreed that it was by no means clear how to proceed beyond menu selection, but thought it impractical to attempt to train salesmen in a programming language.

Professor Page noted that the combinatorial possibilities in the simulated environment could become very large indeed, and to exhaustively test the environments would require complex analysis into equivalent classes that might easily be outside the ability of the person testing.

Professor McKeeman felt there was no chance that the testing could be exhaustive. The objective was simply to detect the first few hundred errors so that the end-user does not see them. All subsequent errors would be fixed at the site when discovered.

Professor Page drew attention to the fact that the 301st error might be the one which gave away all ten thousand dollars. Agreeing with this, Professor McKeeman pointed out that things were worse than that. One suggestion made was that a machine be given to a student who would spend all summer trying to 'break' it. This frightened the manufacturer. They were not at all concerned that these devices could be tricked, but much more concerned that someone knew how. The mystery of the device was part of its protection. That the machine might make mistakes was fairly low on the list of problems. Banks do not mind money being stolen - they are insured against that - what they are worried about is customers withdrawing their money, it's gone then. They are very concerned with their image.

Professor Randell asked Professor McKeeman to enlarge on his point about the computer not being suitable for interpretation. He replied that some microprogrammable machines make good interpreters even where the instruction unpacking is left to the microcode. It appears that if you are going to compile into some engineering-chosen opcode set you are already in difficulties. Machines designed to be interpreters fit better with the current economics which optimise for space rather than speed.

Professor Randell requested more information on the student project implementing part of this system.

About 40 students, mostly undergraduates who were quite good programmers, were formed into three separate groups and had ten weeks to complete the project. The first stage was to write on the Burroughs B5000 an emulator and assembler for the PDP11. Secondly, in that emulator they had to specify a complete banking machine.

All this had to be done in the sense of the software hut where the students had to sell their product to their fellows. This worked very well at Toronto but very badly at Santa Cruz, where the students resented the competitive aspects.

At the end the students had to assess what they had learnt.

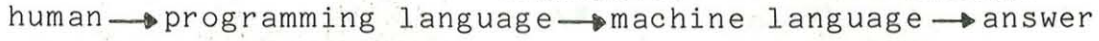
STACK COMPUTERS

Introduction

Professor McKeeman mentioned that his first involvement with stack computers was as a student at Stanford at the time of the Burroughs B5000 when he was sent to the Burroughs plant to learn about their machines. Since then the B5500, 5700 and 6700 have continued the development of stack hardware, becoming increasingly complex, and it was suggested that the B6700 was near the practical limit of complexity for such mechanisms.

In contrast, what follows is an explanation of the simple ideas about stack computers, since one by one these ideas are easy to understand, but all together comprehension becomes difficult.

The first task is to understand what problems are being solved; generally there are questions to which we require the answers. There are many ways to do this, and the way we have chosen is:



Within this model there are alternatives available, and we have a choice of machine language and programming language.

When evaluating a machine design it is necessary to take account of many factors along the entire path from question to answer; the effort to express a problem in a programming language, the time to compile it, then speed of execution etc. Stack computers have their main payoff in programming time and compilation time.

Arithmetic Evaluation Stack (S)

Although this is the simplest type of stack to understand, there are still several design choices which can be made. For example we may choose a mixed stack, where all variable types are manipulated in a single stack; this is wasteful of space, since floating point numbers normally occupy more space than integers, and a large operator set is needed to cover operations between different types such as real add, integer add etc.

Another possibility is to have a different stack for each type; space wastage is avoided, but many operators are still needed, as is some means of transferring values between stacks.

A third method is to have a single stack, but to tag each item in it with its type information as Burroughs does. With this method space is again wasted, since room must be left for tag field, but the operator set can be much smaller - a single add operator can take care of type conversion and checking.

Control Stack (Sc)

This type of stack is concerned with saving and restoring the program counter on procedure entry and exit. As an example of the way such a stack can be used, consider a situation which may occur in a compilation when an error is detected. In a typical compiler, the program may be deeply nested in procedure calls when the error is encountered, and a simple recovery method is to read the input stream until a semicolon is detected, then return to a place in the compiler where a semicolon makes sense; this entails many levels of return. There are several more examples of where this situation can occur, and it would be useful if there was a mechanism in one's programming language to allow this. For example a 'return to' or 'return from' statement. An example of how this construct would be used is shown in Figure 1.

The question then is, in terms of stacks and machine architecture, what is needed to achieve this type of behaviour? The control stack, used exclusively for saving return addresses is one possible solution, and the operations needed to manipulate it to achieve various types of call and return are summarised in Figure 2.

```
P : PROCEDURE OPTIONS (MAIN);
    S : PROCEDURE(N);
            /* some code */
        END S;
    R : PROCEDURE(N);
            IF N<0 THEN RETURN FROM Q;
            CALL S(5);
        END R;
    Q : PROCEDURE(N);
            IF N<3 THEN CALL R(N);
            CALL S(10);
        END Q;
        CALL Q(-1);
    END P;
```

Figure 1

	Call Q	Return from Q	Repeat Q
Absolute	Sc←PC PC Q	PC←Sc	PC←Q
Relative	Sc←(P,PC-PRT[P]) PC PRT[Q]	(T,U)—Sc PC PRT[T]+U	PC PRT[Q]
Multi-Level	ditto	T←NIL while T≠Q do (T,U)←Sc (T,U)←Sc PC PRT[T]+U	T←NIL while T≠Q do (T,U)←Sc PC←PRT[Q]

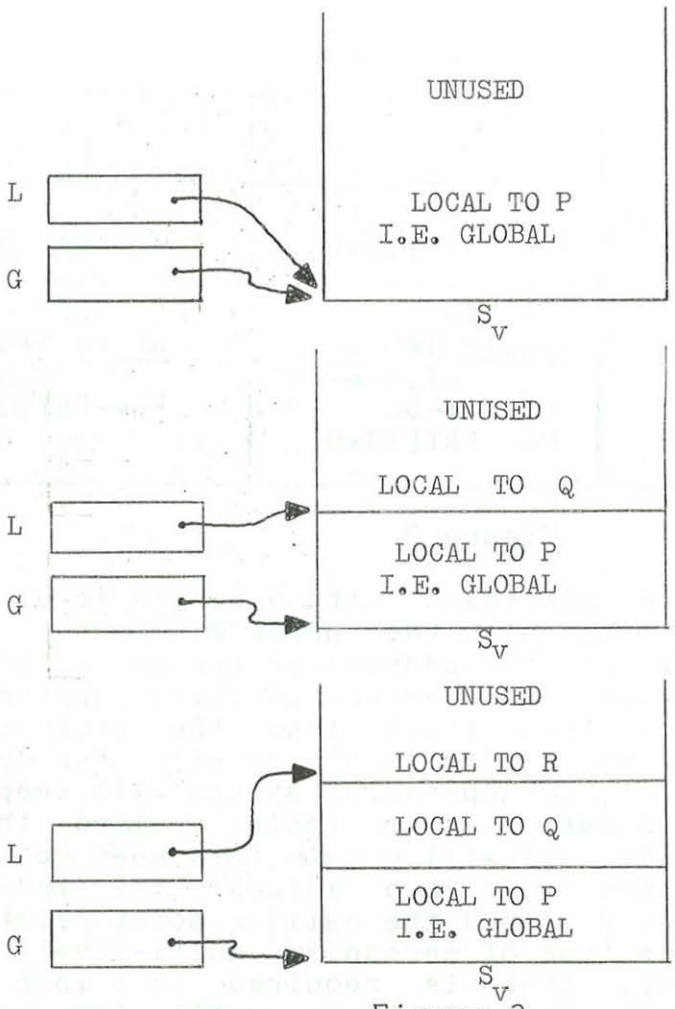
Figure 2

Using absolute addresses with a single level of return, the mechanism is simple - to call, the program counter is pushed onto the control stack, and the address of the called procedure is loaded into the program counter. Return involves popping the saved return address from the control stack into the program counter. This mechanism can be modified slightly to allow for dynamic code relocation. In this case, the operating system will keep procedure entry addresses in an execution time table (called the program reference table, or PRT), and will update this when code is moved. The control stack must then save two values: the index of the calling routine in the PRT, and the calling point relative to its entry point. Using this type of mechanism multi-level return is straightforward, and all that is required is a loop which pops return addresses from the control stack until the current one corresponds to the context we wish to return from. One more pop from the control stack into the program counter will then achieve the desired return.

Local variable stack (Sv)

A third type of stack is the local variable stack, and in terms of the Burroughs machines this has proved to be the most useful. The Algol-like storage allocation mechanism, and addressing relative to global and local base registers (or a display) reduces address sizes and hence code density can be higher.

Figure 3 shows how this stack works. The global register points to the base of the stack for the entire run of the program, whilst the local register points to the base of the local variables of the currently executing procedure, making both globals and locals accessible. Other variables will not be accessible since they are outside the current scope.



P is a program invoked by the operating system. Q and R are non-nested procedures declared within P.

P has called Q

Q has called R. Local variables to Q are temporarily unaccessible.

Figure 3

The operators needed to manage this stack are concerned with scope-entry and scope-exit, and interact to some extent with the call and return mechanisms since the two happen at the same time. Dynamically, the sequence of events must be call . . . . . return, scope-exit, and the actions of the operators are as follows, where P is a program or procedure which calls procedure Q:

Scope-entry	L:= L + V(P)	(In P)
Call	Sc:= PC; PC:=Q	(In P)
Return	PC:= Sc	(In Q)
Scope-exit	L:= L-V(P)	(In P)

(V(P) corresponds to the number of level variables in P).

Thus at scope-entry the local base L is simply incremented in order to protect the local variables of the procedure containing the call, and decremented by a corresponding amount at scope-exit. Therefore at these times the number of local variables in P should be known. Professor Pyle pointed out that since different data types may be of different lengths, what really needs to be known is the space occupied by the variables rather than their number, especially if the language allowed user defined types. Professor McKeeman acknowledged this, and warned against designing a machine with a specific language in mind. If one was to allow user defined types then the kind of scheme which was suggested in the section on

arithmetic evaluation stacks, namely one stack for each type, would clearly to too restrictive.

Professor Whitfield suggested that it would be possible to do the storage allocation for a particular procedure when it was called, rather than having to protect the space occupied by its local variables when this procedure called another. Professor McKeeman replied that the two methods, namely allocation on entry, or protection before a call, were basically equivalent solutions. However, by deferring things for as long as possible, more information is normally available, and in this case one is protecting the thing which is best known, that being the scope in which one is already.

Dr. Chen added that with some interpretive languages such as APL it was not possible to know in advance how many local variables were present in a parcticular procedure.

Marker Stack (Sm)

One further stack, the marker stack can be utilised in conjunction with the local variable stack. On scope-entry, the old value of the local base L is pushed onto Sm before it is incremented, so that all that is needed for scope-exit is to pop this saved value from Sm into L. This implies that the number of locals V(P) is only needed at the time of the call. We then have:

```
Scope-entry  :  Sm:=L;   L:=L+V(P)
Scope-exit   :  L:=Sm
```

The means of addressing variables relative to global and local registers works well if nesting of procedures is denied. When procedures are nested, a procedure has access not only to its local variables but also those of any procedures inside which it is declared. In this case the global/local solution is inadequate, and a more complex mechanism, the display, is necessary. The contents of the display are then used to point to the bases of all scopes to which the currently executing procedure has access. Using display, the scope-entry and scope-exit operators are as follows:

```
Scope-entry  :  Sm:=D[J]
                D[J]:=D[K] + V(P)
Scope-exit   :  D[J]:=Sm
```

and figure 4 illustrates a typical situation.

The above mechanisms are only valid if parametric procedures are not allowed. If this is not the case, then a procedure may be passed to the body of another procedure which is not accessible to the calling procedure. In this case all of the display (except D[0]) may have to be changed, which entails saving all the changed entries on the marker stack.

The various stacks, registers and program code and their interactions are pictured in Figure 5, and a corresponding picture for the B5000 is shown in Figure 6.

The B5000 has one large stack, plus local and global base registers. Arithmetic evaluation is done at the top of this stack, and contained in it are many links to accomplish the same effect as did the marker stack and control stack. A problem with this type of organisation occurs with procedure parameters which are prepared before the call, but eventually are required to be in the local area for the called procedure. They therefore appear on the stack 'a little early'. One way round this problem is to have a parameter preparation stack in which these variables can be stored until the call, and then copied across to where they are to be used. On the Burroughs machines a lot of complicated manipulation takes place at the time of procedure calls in order to ensure correct return from the call. This is a good example of what with a single stack is complicated, but with several stacks causes no problems.

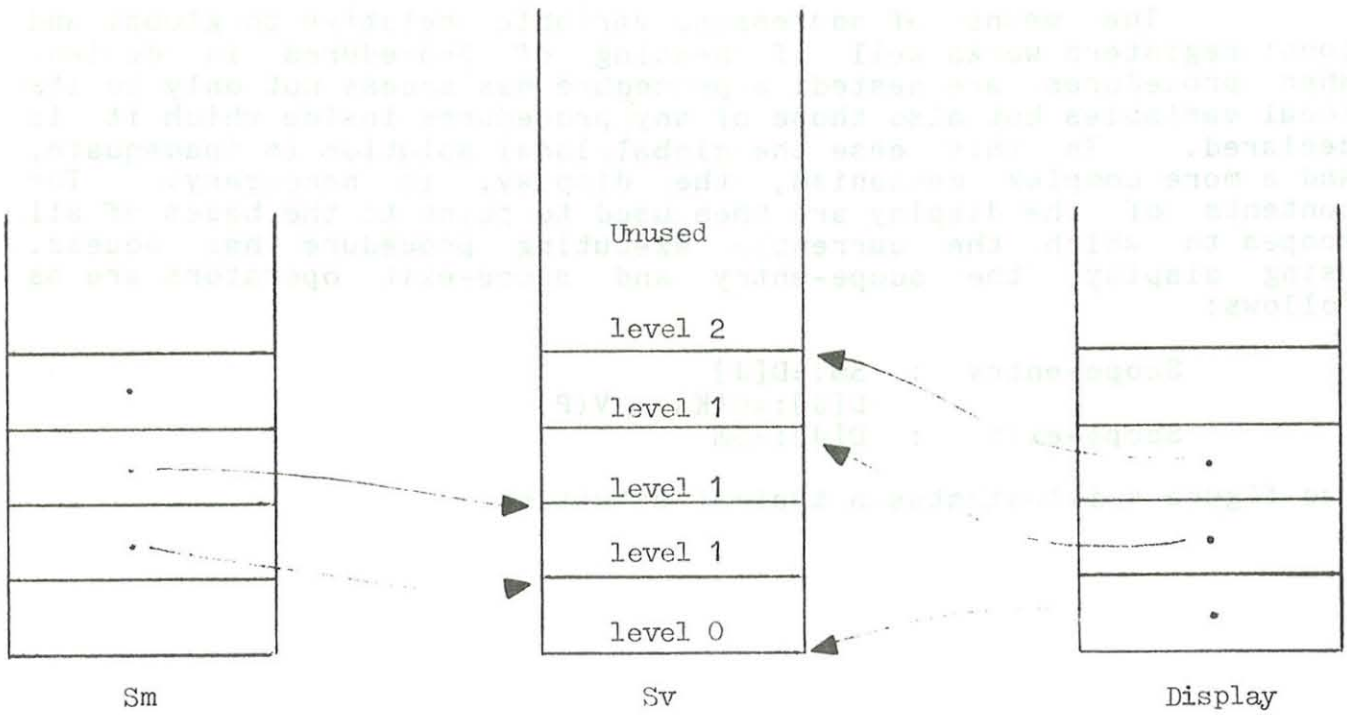


Figure 4

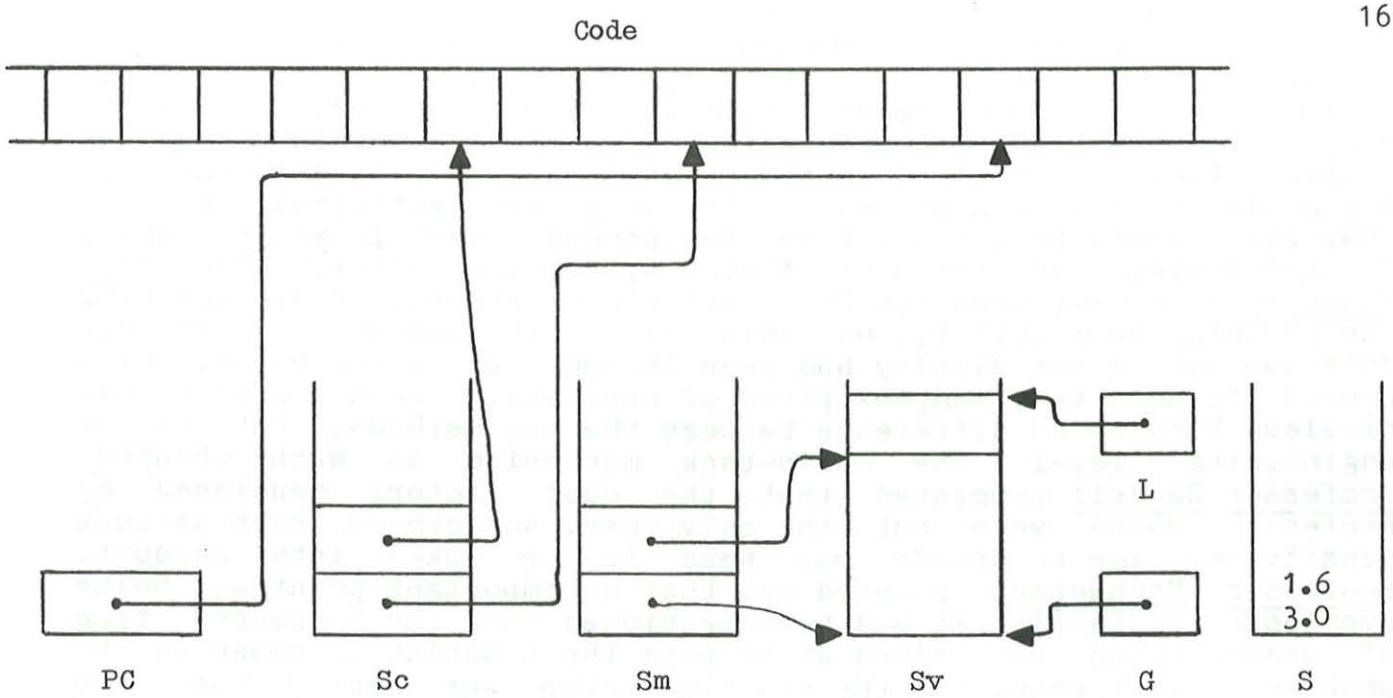


Figure 5

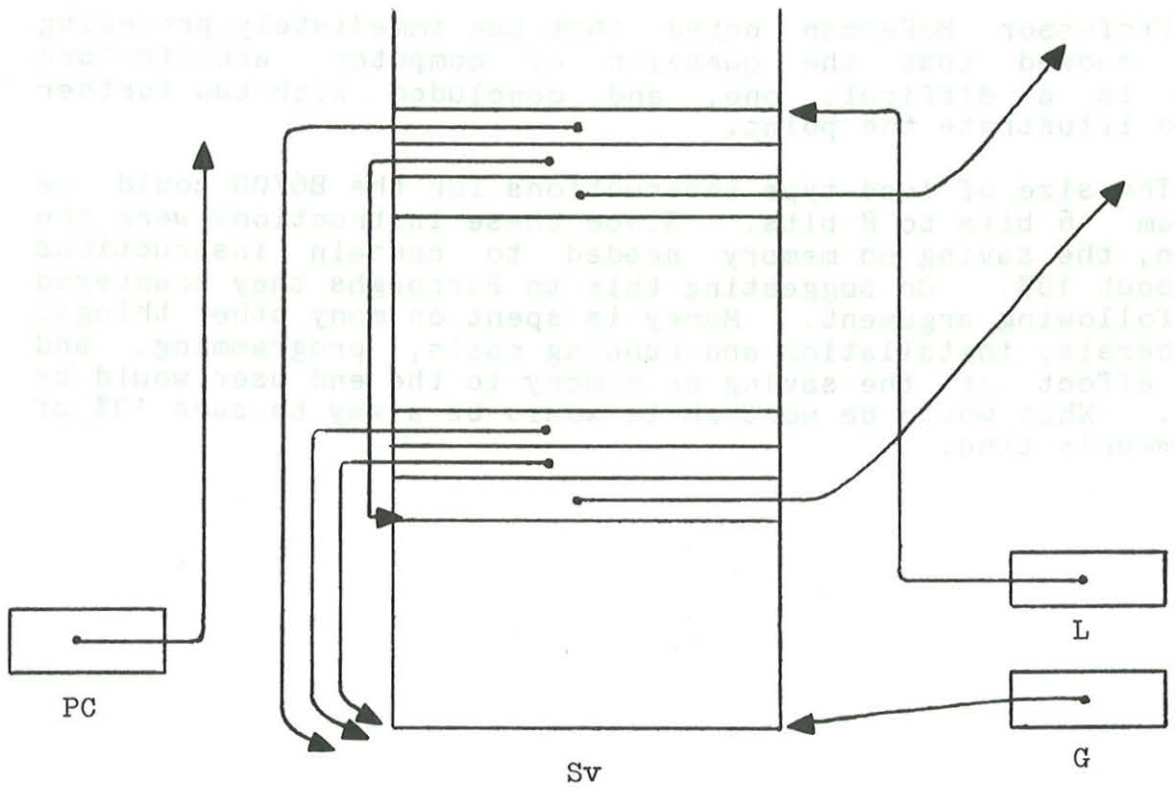


Figure 6

Professor Hoare remarked that, since the cost of compiler writing had been significantly reduced since 1960, in a compiler for a language and hardware as simple as those described, it would be easy to do more address allocation and stack administration at compile time. He also noted that the chain-back method of organising access to non-local variables is very efficient, provided that fast access to global, local and perhaps next-local variables is available, and therefore a display is unnecessary. Professor McKeeman said that when the B5000 was first released it did not have the chaining back ability, but this was later added in software. This was done after display had been implemented on the B6700, which proved to be a very complex piece of hardware. From a user point-of-view there is no difference between the two methods, but on an engineering level the chain-back mechanism is much cheaper. Professor Randell commented that the cost factors mentioned by Professor Hoare were not the only ones, and others, such as code density and time to decode may need to be taken into account. Professor Michaelson pointed out that an important point was being overlooked. Earlier it had been mentioned that the Burroughs type of organisation was aimed at helping the translation phase on the problem solving path, but the run time step was really the more important. Thus the problems of code location and the number of working areas in a roll-in/roll-out system should dominate design decisions rather than time saved at compile time, and certainly at compiler writing time, which is trivial in comparison to the rest. Professor McKeeman expressed doubt about whether compiler writing time should be regarded as trivial, and went on to make two further points. Firstly, when something goes wrong with a program, the time taken by the programmer to find and fix the problem is very important. Secondly, with the addressing mechanism associated with stack organisation, addresses can be much shorter, and a higher code density can be achieved.

Professor McKeeman noted that the immediately preceding discussion showed that the question of computer architecture evaluation is a difficult one, and concluded with two further examples to illustrate the point.

The size of load-type instructions for the B6700 could be halved from 16 bits to 8 bits. Since these instructions were the most common, the saving on memory needed to contain instructions would be about 10%. On suggesting this to Burroughs they countered with the following argument. Money is spent on many other things: cpu, peripherals, installation and running costs, programming, and that the effect of the saving on memory to the end user would be negligible. What would be worthwhile would be a way to save 10% of the programmer's time.

The evaluation question is very hard, even with quantitative cost benefit analysis techniques. A careful evaluation on these lines was done at a particular installation, and it was shown that the machine was costing more than the benefit it produced. When this was explained to the owner of the installation he denied it, since he had forgotten to mention another factor - the fact that he allowed customers to tour his computer centre, and he received much business that way. He knew that the computer was worthwhile, but was unable to express it quantitatively.

Reference Introduction to Computer Architecture, H. Stone (editor). Chapter 7.

A SIMPLE COMPUTER

1. Introduction

Professor McKeeman began his third talk by recounting some early teaching experiences in a course on machine structure and computer organisation which was based upon the IBM 360 computer. He observed that students unfortunately formed the impression that the IBM 360 design was the solution to the problems of machine design. In a subsequent course where students were to design their own machines a disproportional number of 360-like designs emerged. Using the HP2116 computer in this course rather than the IBM 360 simply produced HP2116 evangelists rather than IBM 360 advocates. The simple computer to be described resulted from such experiences; it is sufficiently powerful to give the basic insights into computer architecture but is unlikely to be mistaken to be a pre-ordained solution to all problems therein. It soon becomes clear that it is deficient on both economic and engineering grounds when these issues arise later in the course.

The objectives of the course are

1. To introduce the subject of machine organisation (after students have acquired considerable programming experience - a course on compilers is a prerequisite). This course not only introduces students to their first machines but involves them in designing their own machines.

2. To avoid exposing them too early to "authoritative" designs such as the IBM 360 or HP2116 computers.

3. To introduce the notion of "levels of control". The idea that translation from a high level language to machine language has to be a single step can be avoided by introducing microprogramming from the outset. In this way students immediately see the existence of intermediate levels over which choice can be exercised during the design stage.

4. To introduce alternative CPU organisations. No initial committment is made to zero, one, two or three address or indeed any other form of organisation. Consideration of several such alternatives is included in the course.

To cover this variety, the initial designs involved software implementation - an implementation in hardware follows later in the course.

2. A Simple Microprogrammable Computer

These early designs utilise a simple microprogrammable computer consisting of a number of independent units each with one or more input ports and one or more output ports. To simplify the design each input port is a general register which stores the last data value transmitted to it. The output ports have values which are a function only of the input port values and the state of the unit itself. The sequence of events between setting an input port value and the first inspection of an affected output port value is arranged to have sufficient delay to ensure that the output value is ready.

The units comprising this computer include general registers, special registers, memories and computational units. The output ports of all of these units are connected by a common bus to the input ports of the units though at a given moment in time data may only be transmitted from one output port to one input port.

Each unit and where necessary its input and output ports are named as shown in Figure 1. The bus and unless otherwise stated, the ports are 16 bits wide. Where ports are less than 16 bits the rightmost (least significant) bits of the bus are used, all others being set to zero. Apart from connections to LIT and MUPC which are explained later all data flow is accomplished by a sequence of bus transfers moving data from one of the sixteen output ports (0 to F) shown on the left of figure 1 to one of the input ports which are illustrated on the right of figure 1.

The registers R0 to R3 are general purpose registers. The output port value of one of these registers is equal to the value most recently transmitted to its input port.

IN and OUT are 8-bit registers used for input and output to an external device such as a teletype. The value of IN is the EBCDIC code of a character sent from the input device; a side effect of inspecting the value of IN is to transmit the next character from the input device to IN. Each EBCDIC character code sent to OUT is transmitted to the output device.

The 4 bit register LIT can only be set to a value obtained directly from a microprogram instruction. Unlike other units then its input port is not connected to the bus.

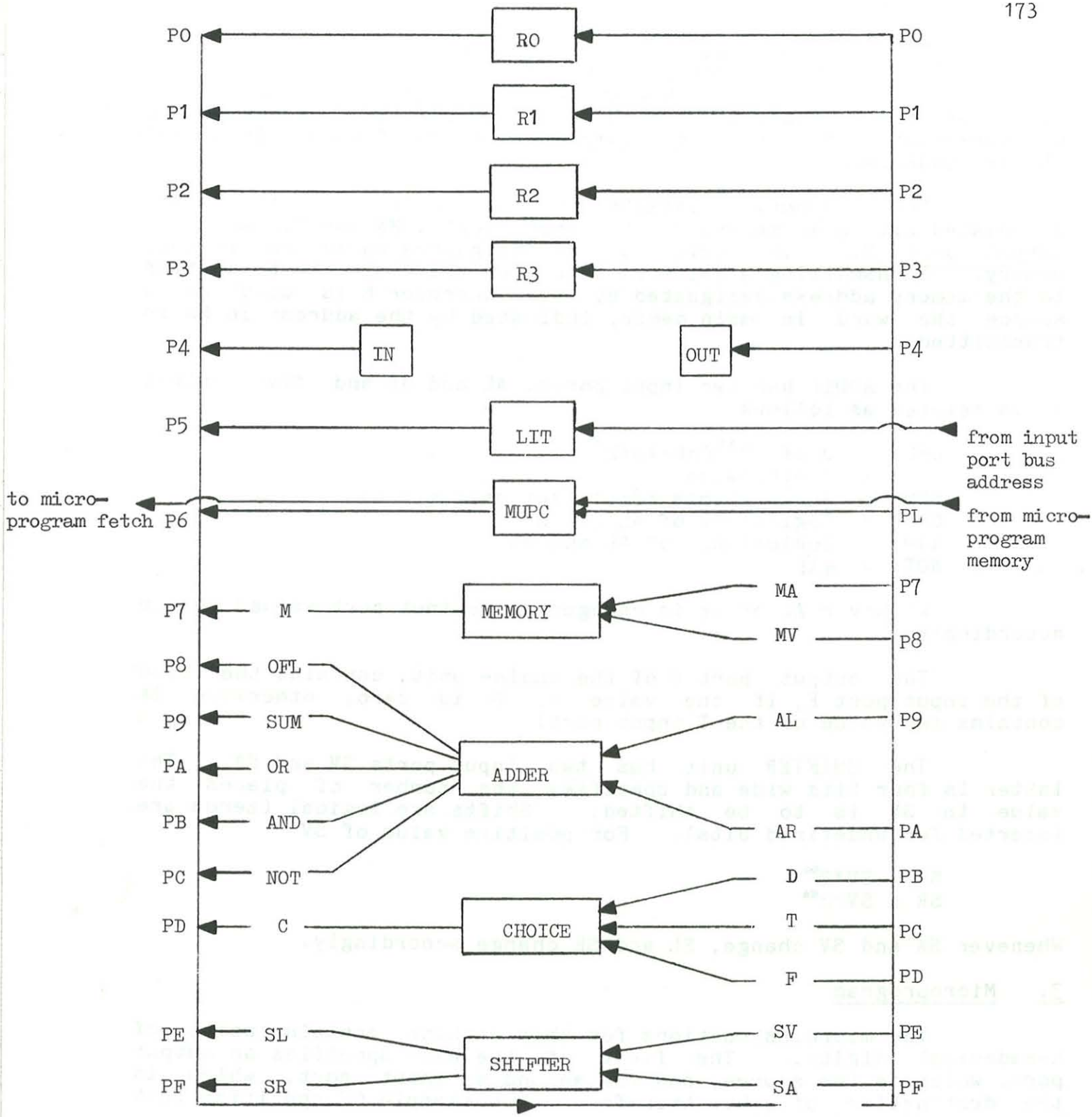


Figure 1. Bus Port Assignments

MUPC is the microprogram counter. It contains the address in the microprogram of the next microinstruction. It is automatically set before the beginning of a microinstruction fetch cycle of the computer. If MUPC is explicitly changed by a microinstruction the next microinstruction is fetched from the changed address.

Main memory consists of up to 2^{16} 16-bit words. Associated with main memory are two input ports, MA and MV and one output port M. The value of MA designates an address in main memory. Transmitting a value to MV causes this value to be written to the memory address designated by MA. Whenever M is used as a source the word in main memory indicated by the address in MA is transmitted.

The ADDER has two input ports, AL and AR and five output ports related as follows

OFL = 0 if $-2^{15} \leqslant AL+AR < 2^{15}$
= 1 otherwise
SUM = AL+AR (two's complement arithmetic)
OR = logical OR of AL and AR
AND = logical AND of AL and AR
NOT = $\neg$AL

Whenever AL or AR is changed all output port values change accordingly.

The output port C of the choice unit, contains the value of the input port F, if the value of D is zero; otherwise it contains the value of the T input port.

The SHIFTER unit has two input ports SV and SA. The latter is four bits wide and specifies the number of places the value in SV is to be shifted. Shifts are logical (zeros are inserted for undefined bits). For positive value of SV

$SL = SV*2^{SA}$
$SR = SV/2^{SA}$

Whenever SA and SV change, SL and SR change accordingly.

3. Microprogram

The microinstructions for this machine contain pairs of hexadecimal digits. The first of the pair specifies an output port, which is the source, and the second an input port, which is the destination of a bus transfer. For example 07 specifies that RO is transferred to MA.

The one exception to this rule arises in the case of the LIT unit. When the destination is 5 the first hexadecimal digit is the constant to be transmitted to the input port of the LIT unit.

A microprogram is a sequence of microinstructions stored in a separate memory (not shown in Figure 1). The mechanism that fetches a microinstruction from the address specified by the MUPC register and causes it to be obeyed constitutes the hard-wired part

of the computer.

Each instruction contains a third field of eight bits in addition to the two hexadecimal digits. This eight bit field specifies the address of the next microinstruction to be executed. During the fetch part of the fetch-execute cycle for a microinstruction the contents of this 8-bit field are transferred to MUPC so defining the address of the next microinstruction. The microprogram memory then is fixed at 256 16-bit words.

Commenting on this computer, Professor McKeeman regretted the pervasiveness of decision to make the bus 16 bits wide but there seemed to be no way of incorporating truly variable length operations sufficiently simply to allow it to be introduced at the outset in a first introduction to computer architecture.

As a simple illustration of the use of this computer, the microprogram for the fetch cycle for a single address computer is shown in Figure 2. The instruction format of the one address computer is assumed to be, four high order bits representing the operation code and twelve low order bits representing an address RO is used to simulate the program counter (PC) and R1 the address register (ADR) of the simulated computer.

Microprogram memory address	Micro-instruction	Symbolic form	
10	0711	PC→MA	Prepare memory access
11	1512	1→LIT	Increment program counter
12	5913	LIT→AL	
13	0A14	PC→AR	
14	9015	SUM→PC	
15	7316	M→R3	Fetch Single address instruction
16	3E17	R3→SV	Isolate address field
17	4518	4→LIT	
18	5F19	LIT→SA	
19	EE1A	SL→SV	
1A	F11B	SR→ADR	
1B	C51C	12→LIT	Switch on opcode
1C	5F1D	LIT→SA	
1D	3E1E	R3→SV	
1E	F61F	SR→MUPC	

Figure 2. The fetch cycle of a one address machine

The symbolic form of the instructions shown in Figure 2 omits the next instruction address. In the first line the program counter is sent to the memory address input port preparatory to fetching the instruction from main memory. The next four instructions increment the program counter by 1 then the instruction is brought from memory into the scratch register R3. The next five instructions take the instruction from R3 shift it left and then right four places to strip off the operation code and leave the remaining address in the address register. The final four

instructions take the instruction from R3 shift it right twelve places and send the operation code to the microprogram counter. The final instruction effects a multi-way branch to microprogram memory locations 00 to 0F according to the operation code.

Clearly the instructions of the Fetch/Execute program may reside anywhere in the microprogram memory (other than location 00 to 0F) and it has been assumed that they reside in locations 10 to 1E for the purpose of specifying the microinstructions fully in the second column of Figure 2. The microprogram memory layout so far specified is shown in Figure 3.

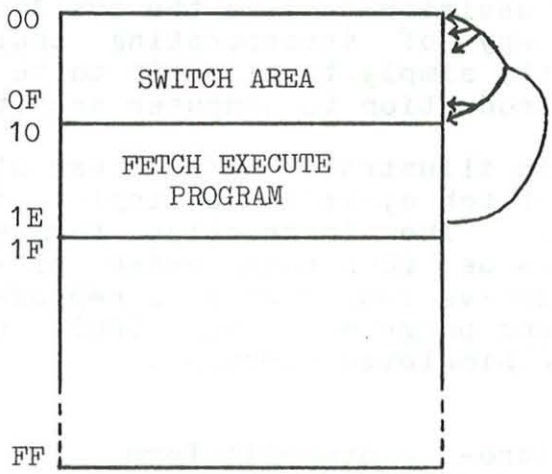


Figure 3 Microprogram Memory Layout

Each microinstruction in the switch area is the initial microinstruction of the execute cycle for the corresponding operation of the one address machine. This initial microinstruction designates as its successor the first of a sequence of instructions (in the unallocated region 1F to FF) which complete the execute cycle of the one address machine instruction. While the memory layout becomes somewhat bizarre it is not difficult to understand and is part of the price paid in keeping the underlying machine simple.

4. Teaching

Professor McKeeman next described the way in which the course using this computer develops.

First of all a simple single address computer with 4-bit opcode and 12 bit address is described and students write a few short simple programs for it; examples might be a program for generating the Fibonacci numbers or for performing bubble sort. These are punched on cards and fed to the one address "machine" which is emulated on the microprogrammed computer. (The latter is in fact an XPL program running on hardware which is more readily available.) The one address machine provides them with about one page of output in the form of an instruction by instruction trace of the execution of this first assembly language program. (Strictly speaking their first machine language program since the coding is done in hexadecimal.) They then see the states of all the registers

during the execution of a few instructions. Decoding this output becomes dull very rapidly but once the students have understood the behaviour of the one address machine they are exposed to the microprogram implementing the one address machine. They are helped in understanding it by running their programs once again but with a different set of trace mechanisms which trace the action of the microcomputer displaying the bus cycle. This gives more than a page of output and since interest in it wanes quickly it is limited to about 100 lines. Now having seen the macro machine and the micro machine operating they have seen most of the mechanisms and can now design a machine, usually a stack machine, and reprogram the microcomputer to implement it. Alternatively they might add an "execute" instruction to the one address machine or perform some similar exercise. In this way students get the idea of adding opcodes, changing opcodes, modifying the whole architecture of the machine.

The simple computer provided is sufficiently general that these things can all be accomplished. It is soon apparent to students that it is far less than ideal. Its main virtues are that it works in a straightforward fashion and that an engineer has never seen it - which means that its use is not constrained by a large number of subtle features (perhaps based on what core technology required in 1956). The machine was designed to be understood, economics was not a consideration. At about the time that the students become disenchanted with the microcomputer they are set the task of designing their own microcomputer. At this stage they are made aware of the fact that the microcomputer is in fact merely represented by an XPL program which is not sacrosanct and they are then free to design a better micromachine than that provided. At last then they are on the way to studying computer architecture.

The first three weeks of the course are taken up by these things. Some of the students may then write an assembler for their microprocessor, using a symbolic form for instructions, taking advantage of what they have learned in the compiler writing course. The main functions of the early part of the course are concerned with getting over the ideas of the fetch execute cycle at different levels and the choices that are available here. Then about a week is spent on exercises such as

1. Design (and implement?) the bus.

Enough background in electronics is available at this stage to allow consideration of how to build the latches on the end of the bus for each of the ports and how to decode the bus addresses. It would be nice to have sixteen lines with a signal on a specific line to denote a specific address but an alternative is to encode addresses on four lines and use 4 x 16 decoders; each latch must then be selected on a particular bit pattern. Yet another alternative is to associate the decoding with the ports.

2. Design the adder unit.

This exercise would stop at the design stage. With ALU chips there is little point in implementing it.

3. Design a 4-bit Polish Machine

This turns out to be painful. The underlying machine has a sixteen bit memory. Once students have considered the implications of packing four instructions per word they are shown the B5000 solution.

4. Make the microcomputer emulate itself

This is a non-trivial exercise and leads to the same issue as the Universal Turing Machine. Can the microcomputer emulate itself emulating itself . . . . . ? By changing the IN and OUT registers one student did emulate a Turing Machine.

5. Emulate the PDP 11 (?!)

Students have seen enough of the PDP11 to realise that it is a very simple and neat computer and they begin to contemplate emulating this in a 256 word microprogram memory. After perhaps an hour it is realised that no amount of packing will suffice and the issue becomes "Can you design a microprogrammable computer of sufficient power to emulate a PDP 11 given a constraint on the size of the microprogram memory to perhaps 100 words?". (Professor McKeeman explained that his students at least are used to questions to which the answer is that there is no answer.)

6. Design a machine with addresses of the form [length, address] where address defines a bit address and length defines the number of bits of the operand field starting at the given address.

While this is logically possible on the given microcomputer it is beyond its capabilities given any reasonable amount of microprogram memory. As in the previous exercise this becomes a matter of designing a suitable microcomputer to support variable length operations.

This course runs for two quarters. Presentation of the simple computer occupies the first two weeks. After this students are taught logical design and they eventually build a computer involving about 600 to 700 backplane wires. They have certain aids such as wirewrap programs.

Professor McKeeman closed by remarking that at least he did not recognise HP2116 or IBM360's among the machines designed by students on this course.

5. Discussion

Professor Heath commented that he liked this approach and the simple computer and the fact that it left out detail which caused loss of momentum in teaching. He cited as an instance the fact that the program counter is set by each microinstruction rather than incremented avoided the lengthy digression into branch instructions which would be necessary at that point in the case of teaching based on say the PDP 11. (Interestingly Professor McKeeman had remarked earlier in an aside that he had in fact done this for a good engineering reason. Ripple carries in the program counter cause the execution of microprogram instructions on a slow, quick, slow, quick, very slow . . . . basis.)

Replying to a question by Professor Dijkstra, Professor McKeeman explained that 2 lecture hours and 2 laboratory hours were allocated each week, so the presentation of the microcomputer took about 4 hours of lectures - the corresponding laboratory periods were occupied by running programs and explaining computer centre procedures. Mr. Givens asked what level of students took the course. The course was intended for 3rd year undergraduates who should have already have taken the compiler writing course as a prerequisit. Professor McKeeman said he believed that no one should be designing computers without understanding what compilers do but a number of students sneaked in without it. Perhaps half of the class were third year students, with a few postgraduate students and a few first and second year students.

Professor Vranesic asked if it were the case that engineers who knew nothing of compiler writing should not design machines, then was it reasonable that compiler writers who knew nothing of engineering should do so. Professor McKeeman's initial response was "Certainly, because it's fun." More seriously he replied that of course programmers in general have not the first idea of how to convert things into logic and he then commented on his experience with Burroughs where the design teams include both programmers and engineers. The engineers with their different background frequently instinctively recognised things that would compromise the total design. As an example he cited the B6700 design where the programmers lead the design team and the engineers sincerely attempted to implement their design. Subsequently it became clear that the programmers had not appreciated fully some of the engineering consequences of their decisions, some of which impaired rather than improved the efficiency of the machine overall. Fortunately the chief engineer on the design team taught himself a lot about programming during the project so that he could understand what it was that the programmers were about. Subsequently in the design of the B1700 the pendulum swung back and an engineer was in charge. Here the programmers were merely advisers. In this case the programmers' advice was that the early versions of the machine could not be programmed. The engineers' reply was "Of course it can be programmed. It can emulate a Turing Machine." Finally the chief engineer agreed to program the machine for a few weeks and the final version of the B1700 reflects the results of his agreeing to train himself in programming. In both cases it was the engineer who of course really did the detailed design but the combination of programming and engineering experience was beneficial in both cases. Professor Vranesic then asked if the student, at the end of the course, was aware of the engineering constraints to which Professor McKeeman replied that they were fortunate at Santa Cruz to have a very skilled computer engineer on the staff who supervised the later laboratory work and introduced students to those aspects of design to which a good electrical engineering background contributes. He, for example, will not permit them to build the simple microcomputer - it is too expensive for what it does - it has too many parts. He discusses an electronically much simpler machine but which logically is more complex. It involves 600 - 700 backplane wires rather than the 1800 involved in building the simple computer. In addition, students eventually obtained experience of interrupt programming on a PDP 11; they program both tbe B5000 and the IBM360 so they do see a variety of real machines (in the case of the B5000 even at the wiring diagram level).

The response to a question by Professor Whitfield on numbers of staff and students involved was that the class never contained more than 20 students so that one member of teaching staff could handle it. Availability of staff interested in teaching different aspects of the course resulted in it being split between the two. Professor Whitfield commented that it seemed students must spend a great deal more time on this course than the contact hours allocated to it. Professor McKeeman explained that students at Santa Cruz normally take three courses simultaneously which implies about 12 to 15 contact hours; they are however expected to spend twice as much time outside of class making about 40 to 45 hours of study per week. However students do complain that the computing courses demand more of them in terms of time than other courses; the level of effort was indicated by the fact that the students threw a champagne party on the day that the machine they had built worked. They were very proud of that achievement.

In the ensuing discussion about the organisation of the group project resulting in the construction of the machine it emerged that the management was largey controlled by the engineering staff member though the students perhaps would not agree. He vetted the design carefully before allowing any implementation and prevented really bad decisions which would have caused trouble. The students divided into four groups, the memory section, the I/O section, the fetch-execute section and the combinatorics section. The groups were rather arbitrarily composed and were in any event dynamic.

Reference A Simple Computer : SIGMICRO Newsletter, October 1974.