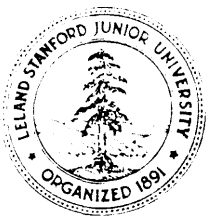
DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY · STANFORD, CA 94305




SU-SEL 77-030

# INTERPRETIVE MACHINES

by

John K. Iliffe

June 1977

## Technical Report No. 149

The work described herein was supported in part by the Joint Services Electronics Program under Contract No. N00014-75-0601. The lectures also form part of a course on "The Microprocessor and its Application" held at the University College Swansea under the auspices of the Informatics Training Group of the E.E.C. in September 1977.

SU-SEL 77-030

# INTERPRETIVE MACHINES

by

John K. Iliffe

June 1977

Technical Report No. 149

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

The work described herein was supported in part by the Joint Services Electronics Program under Contract No. N00014-75-0601. The lectures also form part of a course on "The Microprocessor and its Application" held at the University College Swansea under the auspices of the Informatics Training Croup of the E.E.C. in September 1977.

SU-SEL 77-030

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

Technical Report No. 149

June 1977

# INTERPRETIVE MACHINES

by

John K. Iliffe

## ABSTRACT

These lectures survey attempts to apply computers directly to high level languages using microprogrammed interpreters. The motivation for such work is to achieve language implementations that are more effective in some measure of translation, execution or response to the user than would otherwise be obtained. The implied comparison is with the established technique of compiling into a fixed general-purpose machine code prior to execution. It is argued that while substantial benefits can be expected from microprogramming it does not represent the best approach to design when the contributing factors are analysed in a general system context, that is to say when wide performance range, multiple source language, and stringent security requirements have to be satisfied. An alternative is suggested, using a combination of interpretation and a primitive instruction set and providing security at the microprogram level.

The work described herein was supported in part by the Joint Services Electronics Program under Contract No. N00014-75-0601. The lectures also form part of a course on "The Microprocessor and its Application" held at the University College Swansea under the auspices of the Informatics Training Group of the E.E.C. in September 1977.

# INTERPRETIVE MACHINES

J. K. Iliffe

International Computers Limited

These lectures survey attempts to apply computers directly to high level languages using microprogrammed interpreters. The motivation for such work is to achieve language implementations that are more effective in some measure of translation, execution or response to the user than would otherwise be obtained. The implied comparison is with the established technique of compiling into a fixed general-purpose machine code prior to execution. It is argued that while substantial benefits can be expected from microprogramming it does not represent the best approach to design when the contributing factors are analysed in a general system context, that is to say when wide performance range, multiple source language, and stringent security requirements have to be satisfied. An alternative is suggested, using a combination of interpretation and a primitive instruction set and providing security at the microprogram level.

The early lectures review the history and terminology of microprogrammable machines. Knowledge of conventional practice is assumed. Readers already experienced in microprogramming should skip rapidly to Lecture 3.

## 1 MICROINSTRUCTION DESIGN

If we abandon the conventional machine code (at least temporarily) as a means of defining the computer's function set it is necessary to fall back on the next level of description, i.e. the microcode. A very extensive literature has grown up around that subject in recent years, but I think it is true to say that no commonly accepted theory or principles have emerged: that is the consequence of rapid changes in the process of manufacturing logical devices which force a continual revision of the economics of design. In the introductory lectures we shall study the evolution of microprogrammed machines, but one can do little more than present a collection of techniques. For detailed study of application to machine language interpretation the student is referred to Husson (1970), where an extensive bibliography to

1968 will be found, and to Boulaye (1971), for a shorter survey of techniques. In the following notes I can do no more than provide an outline of design principles and introduce terminology.

The branch of technology that enables a raw microprocessor to interpret a given order code is termed 'microsystem design'. If one machine is to interpret one order code it is a very localised affair. If several machines must imitate two or three order codes the need for standard procedures and documentation arises: in the major application areas this is treated very much as an extension of the logic design. Tucker (1967) and Husson have written informatively on that aspect of microsystems. However, high level languages are not nearly as well defined as machine codes, they are generally more complex, subject to greater variation, and outside the control of any one laboratory.. A survey by Rosin highlights some of the difficulties involved, Rosin (1969). We shall return to that subject in the last lecture, showing how it affects machine design. For the time being, let us recall how a microprogrammed machine handles the interpretation of a single 'target instruction set' or 'machine code'.

The first application of microprogramming as a formal technique is generally attributed to the designers of EDSAC-2 at Cambridge University, Wilkes (1958). It is a systematic way of controlling the flow of signals through the data paths of a processing unit, each path, or in some cases each function of the processor, being determined by a bit in a microinstruction. If we regard the state of the processor as defined by the assembly of registers and control flip-flops, then a microinstruction determines a simple transition from one state to another. The attraction of the technique is that transformations of any complexity can be composed by applying a sequence of microinstructions: the limitations imposed by ad hoc control logic, which are apparent in the areas of machine definition and construction, are greatly reduced. At a time when relatively complex target instructions are thought to be the key to greater machine efficiency, the introduction of microinstructions obviously has great attraction.

The source of microinstructions is a store, which will be called the control memory in the present context. A single bit in the microinstruction can control the transmission of an entire field from one register along several parallel paths in one processor 'cycle'; another bit, or group of bits, will select a destination register and field. It is fairly easy to evolve a requirement for fifty or more bits in the microinstruction to control the possible data paths in the processor.

The second requirement of the microinstruction is to determine its successor. Application of a sequencing rule determines the string of actions carried out by the processor which, whenproperly defined, will interpret a target instruction. One of the simplest

ways of sequencing is to place the next microinstruction address in the one currently being obeyed. To achieve conditional branching effects it is necessary to use the state of the processing logic in the calculation of at least part of the next address. The elements of the machine can be visualised as in Figure 1. The machine operates in three steps; i.e.:

1. Access control memory using the microinstruction address.
2. Use the microinstruction to control the state transition of the processor logic
3. Use microinstruction digits and the result of step 2 to determine the next microinstruction address.

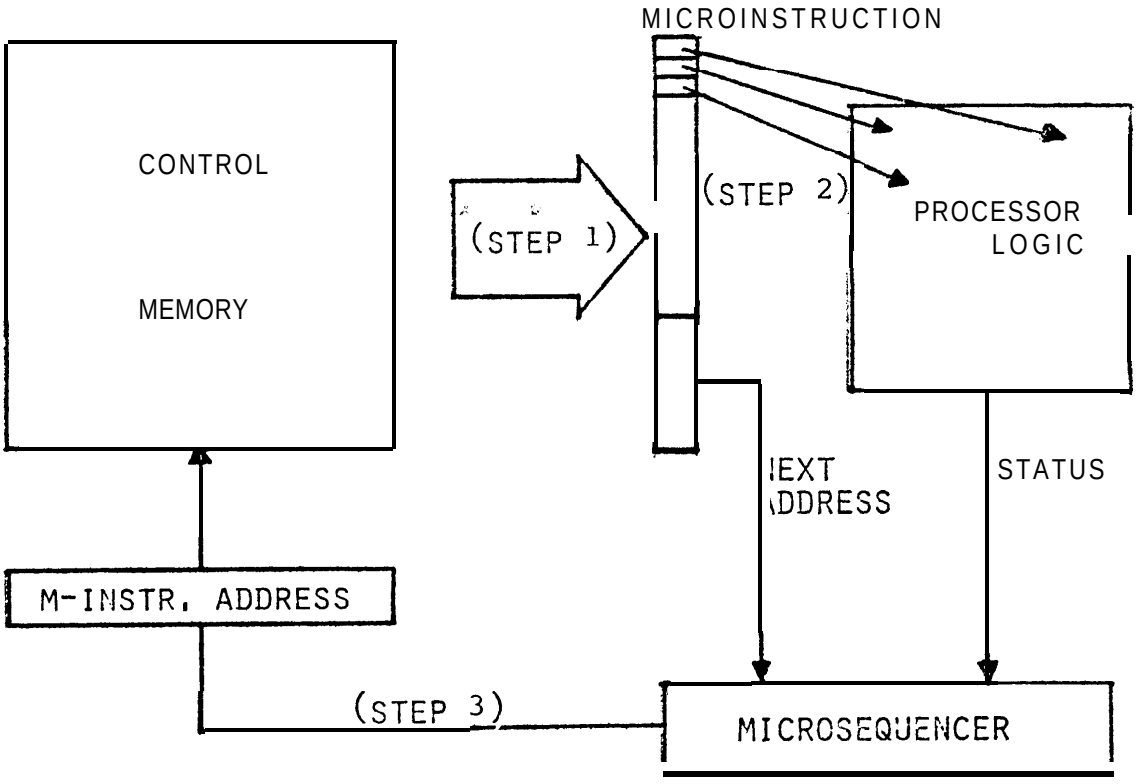



Figure 1: Microprogram Control

The development of microprogrammable machines from the above principle of design leads to great elaboration of detail, the main considerations being (a) optimising the use of control memory, (b) achieving balanced timing of control memory and processor logic, and (c) organising the registers and data paths of the processor to suit the class of target machines of interest. I shall discuss each aspect of design, giving examples from some of the earlier microprogrammed machines.

## 1.1 Minimising the Cost of Control Memory

Exploitation of microprogramming was not widespread until suitable techniques for loading and manufacturing control memory had been developed. Such techniques are discussed by Husson (Chapter 5), where it can be seen that the predominant forms of construction allowed microinstructions to be read but not written under program control. That is clearly sufficient for a well defined and fixed instruction set. The later development of semiconductor control memories with write capability has been the main stimulus to further research in microprogram application. With all memories, however, the main design requirement is to deliver the information required at the right time and in as few bits as possible.

Considerations of space lead to various forms of microinstruction coding. The form in which a single microinstruction bit controls a unique processor gate (or data path) is termed direct control. If we can find sets of mutually exclusive control signals, such that not more than one is activated in a given cycle, it is possible to encode them: a field of K bits will activate one of $2^{K-1}$ control lines, or none at all. That is obviously the case when one of, say, 8 registers can be gated to one input of an adder. The same technique is used in machine code design. It is illustrated below by the structure of the IBM 360/30 microinstruction and by most of the 'first generation' microcodes, all of which may be said to use encoded control, the individual fields controlling microorders.

Three other common forms of coding deserve mention. In bit-steering the particular control lines activated by a microorder (or bit) are determined by another field of the microinstruction. The second field directs the first to one or another set of control lines; it is appropriate when the processor logic can be partitioned into sections that do not require activation on every cycle (and can to some degree proceed in parallel). It has been used in combination with other techniques, for example in the RCA Spectra 70/45, Honeywell 4200 and IBM 360/25. Carried to the extreme, the microinstruction ends up as a function group and a number of operand fields, which would be difficult to distinguish at first sight from a conventional machine code.

The second technique derives from the observation that over many sequences of microinstructions the values of certain control lines will remain constant, therefore they can be set in advance and taken as an implicit extension of the microinstruction. That technique will be referred to as preset control. It applies, for example, if particular carry or shift paths are fixed in advance, or if one of several possible register sets is being used.

Finally, it is easy to see that all $2^{100}$ versions of a 100-bit direct control microinstruction will not be used, and instead of attempting to encode individual fields it would be possible to list all the distinct microinstructions in a particular application and select those required by indexing a store containing the list. For example, in a particular application there may be less than 1024 distinct microinstructions. In that case a 2000 word microprogram can be compressed into 20 000 bits, a saving of 90%. All that is required is that the fully encoded microinstruction index another store 100 bits wide containing the 1024 fully decoded instructions (the second store is called the nanostore). The net saving in storage space is thus 40%.

It is more like that some of the fields of the microinstruction will be fully used, leaving a residual field to be handled in the above way. The Nanodata QM-1 machine, Rosin et al (1972), provides an illustration. The 16 bit microinstruction is loaded into one of the microregisters, a six bit field is then used to select a 342-bit nanoinstruction. The latter can use the remaining ten microinstruction bits as operand selectors, so it is appropriate to regard them as a form of preset nanocontrol (Figure 2). At this point the designer faces the same set of choices at nanomachine level as we have already discussed in connection with micromachines. He could use direct control: in fact, QM-1 does not, but obeys a far more elaborate sequence of nanoorders. The reader is referred to the literature for details.

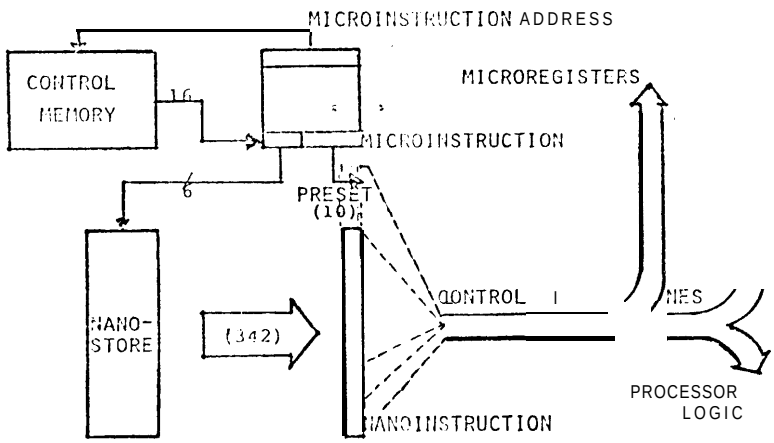



Figure 2: Nanoprogram Control

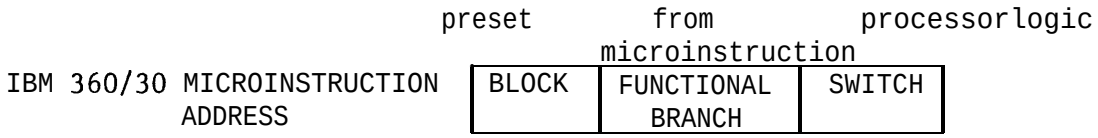
## 1.2 Timing and Control Considerations

It will be shown later that interpreting one of the common target instructions takes approximately 20 microorders and two main memroy cycles. If a premium is placed on memory utilisation it follows that the effective microorder rate must be ten times that of main memory: to achieve that the early machines use a horizontal or multi-order microinstruction that activates between five and ten processor paths in parallel. The microinstruction rate is synchronised to $\frac{1}{2}$ or $\frac{1}{3}$ the memory cycle time so that a 1.5 μsec core memory would be associated with a 750nsec or 500nsec microinstruction rate. Horizontal coding achieves speed at the expense of generality and ease of programming: in the next lecture we shall introduce a more 'relaxed' form of code in which each microinstruction contains only one or two microorders, which is naturally called vertical control.

The elementary steps of the machine execution cycle have already been indicated. If no overlap is attempted then the major components--control memory and processor--are alternately idle while the other completes its task (remember that read-only memories, and even writable semiconductor memories, may require very little time to recover for the next cycle). In order to achieve higher performance it is necessary to use faster and therefore more expensive components, or to overlap the elementary steps. The options are superficially the same as in machine code design. The main differences derive from the fact that micro-programs have been for the most part fixed, comparatively small, and have made extensive use of multiway branch or switch instruc-tions: the alternative of using a sequence of tests to decode a target instruction would simply be too slow.

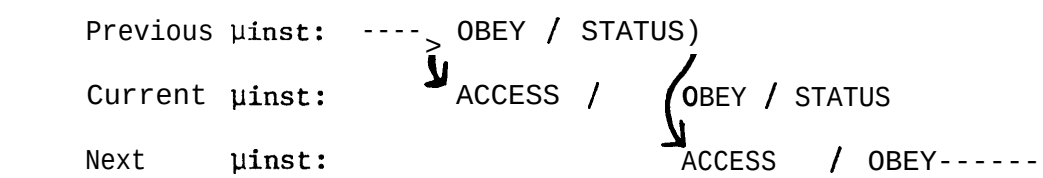
A control memory address is frequently composed from several fields whose values are determined at different points in the machine cycle. The high order fields are normally known first, so the construction of an address reflects a gradual narrowing down of the alternatives until the exact microinstruction can be fetched.

In the IBM 360/Model 30, for example, a block address is found as part of the preset control, not normally affected by the current microinstruction; a functional branch is a field inserted directly from the microinstruction, and a switch is the low-order two-bit field of the control memory address, computed from the processor state. Thus, the successor to any instruction is within the current block of 256 (see diagram) and may be dependent on the outcome of one or two conditions or register values.




We can now see more clearly when the overlap of processor and control memory cycles can be achieved. If the control address is determined by the processor state at the end of the current micro-instruction then although access might be initiated on the basis of block/functional branch fields the final decision has to be delayed until the state of the processor logic is known (the example given above falls into that category).

If the control address is determined by the processor state at the end of the previous instruction, then the control memory can be accessed while obeying the current instruction, e.g.




The timing considerations just described are shared with very much more sophisticated processors: they result from any attempt to overlap one instruction with others and it is easy to see that the more 'changes in direction' in the flow of control the less effective are the overlap arrangements. It is true to say that microprogram is more afflicted by conditional and computed branches than machine language program, for which reason designers are reluctant to throw away the contents of the micropipeline and may ask the coder to deal with various 'run-on' conditions. What this means in practice is that one or two instructions in written sequence after a branch may be obeyed, e.g. in decoding a hypothetical target instruction the microsequence is written:

$m_1$ : Extract function field

$m_2$ : Branch to address + function

$m_3$ : Increment target instruction counter

Here, although the branch $m_2$ is taken, the following microinstruction is still obeyed. It is in avoiding or dealing with such coding peculiarities and in taking account of critical memory or I-O timing constraints that microprogramming differs from conventional coding, or has done so in the past. Luckily, increasing

hardware power has removed many of the characteristics of micro-program from modern machines, perhaps the only positive way in which a microprocessor can be distinguished from a 'mini' is in its dedication to the task of modelling processors rather than users' problems.

## 1.3 Highway and Register Organization

The basic requirements for imitating a given target instruction set are:

(a) arithmetic primitives for composing the arithmetic, logical and addressing functions of the target machine;

(b) memory mapping and resolution compatible with the store structure of the target machine;

(c) imitation of the internal control states, registers and register access requirements of the target machine;

and (d) peripheral interfaces that reflect the formats, status and timing expected by the target machine.

Within this field the degree of dedication varies with the performance/cost objective. Different design teams have gone about the same task in quite different ways: Husson (p414) makes the point that although the IBM 360 and RCA Spectra 70 achieve the same architecture the latter is a much more 'specific' design than the IBM models.

In this subsection I shall illustrate features of microprocessor design referring to the IBM 360/Model 30 which was one of the earliest models of the IBM 360 range and, as it happens, the subject of an early experiment in language oriented design that I shall refer to later. Further details will be found in Boulaye (1971) and Weber (1967).

Figure 3 shows the data paths in the central processor of the IBM 360/Model 30. There are twelve registers, each of one byte. Apart from the main memory address and data buffers (MN and R) no specific allocation of content is made by hardware. The data paths are uniformly 8 bits. The microinstruction is 60 bits long, encoded into the following microorder groups:

| | | |
|---|---|---|
| (i) | Store access: | Fields CM, CN, CU |
| (ii) | Data flow: | 4-bit literal field CK |
| (iii) | ALU control: | CA, CF, CB, CG, CV, CD, CC, CZ |
| (iv) | Sequencing: | CH, CL |
| (v) | Status: | CS |

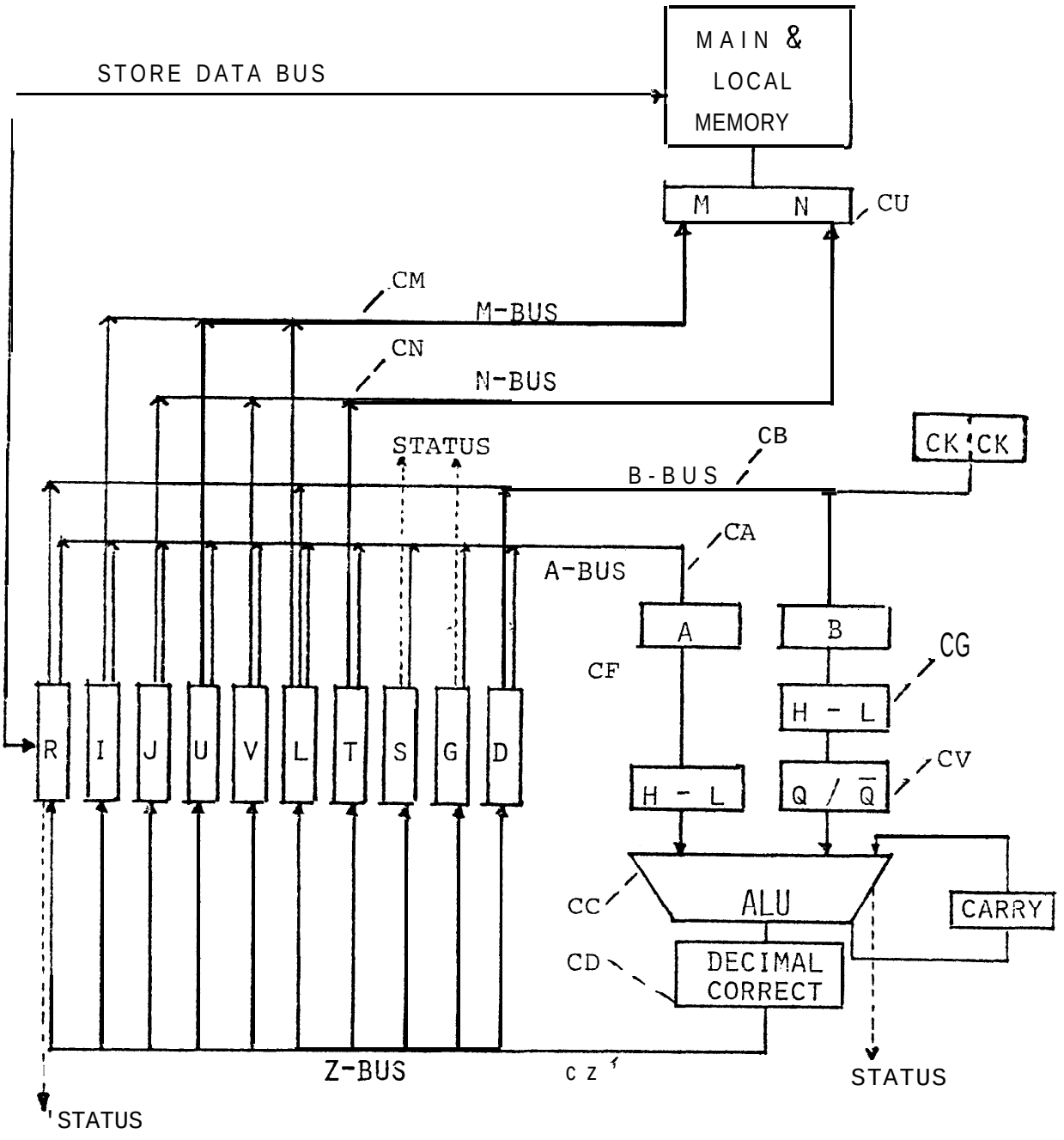



Figure 3: Simplified Data Flow of the IBM 360/Model 30 CPU

For example, under group (i):

CM (3 bits) indicates: No action
Read from address IJ, UV, or LT to R
Regenerate
Write from R

CU (2 bits) selects main or local (register) storage.

Under group (iii):

CA (4 bits) selects one of 10 inputs to the ALU through the A register

CB (2 bits) selects one of R, L, D or the literal CKCK

CC (3 bits) selects the actual ALU function

CF (3 bits) modulates the A-input to ALU, i.e. high digit, low digit, none, low or cross-over

CG (2 bits) modulates the B-input to the ALU

CV (2 bits) selects true, complement or six-correct form of B

CZ (4 bits) gives the destination, one of ten registers.

Thus in one microinstruction, which takes 750nsec, an 8-bit arithmetic or logical operation is carried out, half a main store cycle is controlled, and the next microinstruction is selected. In the next cycle the main store operation must be completed while other operations are carried out.

If we consider the loop of instructions which interprets the target machine code it clearly consists of first fetching the instruction, then looking at the function/format digits and preparing each operand by computing an address and accessing the store when necessary, and then branching to the 'semantic' microsequence that interprets the target function. The instruction will normally terminate by servicing interrupts before proceding to the next in sequence. Elementary IBM 360 instructions take between 15 and 30 μsecs in execution, i.e. 20-40 microinstructions: the large number reflects the fact that any address or arithmetic calculation involving operands of more than 8 bits has to be carried out serially by byte.

In order to achieve higher performance the microregisters and internal data paths must be more closely matched to those of the target machine, and supplementary functional units introduced to minimise the 'mismatch' between the microprocessor and the target system architecture.

## 2. GENERALIZED HOST MACHINES

We have seen some of the ways in which specific features are built into microprogrammable machine to help in modelling particular order codes. However, our main objective is to consider systems at a level removed from machine code, where the target instruction sets can to some extent be chosen to suit the available hardware: in the last lecture we can attempt to answer the question of whether the need for specific adaptation will still arise.

I shall now discuss design generalisations that have been favored in recent years as the result of rapid reduction in the cost of storage and logical devices. In the latter context 'regularity' of hardware is at least as important as circuit or gate count, which is greatly to the benefit of the microprogrammer. I shall refer to the class of processors under discussion as host, machines in order to suggest their role and to avoid undue emphasis on 'microprogram' or 'microprocessor' technology. In practice, the principal use of host machines has been in the form of instruc- set emulators (e.g. IBM 360 imitating the IBM 1401). The design objective of producing a 'universal emulator' became feasible with the introduction of writable control memories. It is clear from the outset that machines capable of imitating any instruction set at competitive speed could not be produced at competitive cost, nevertheless such a machine is invaluable as a vehicle forresearch into computer architectures. The ICL Research Emulator El, Iliffe May (1972), the Standard Computer Corporation MLP-900, Rakocsi (1972), the Stanford University EMMY, Neuhauser (1975), and the Nanodata Corporation QM-1, Rosin, et al (1972), provide examples of generalised facilities, while in the commercial field the Borroughs Corporation B-1700 is particularly interesting from the point of view of memory allocation.

All the machines in this category use vertical instruction coding which allows much greater flexibility in function sequencing than the older horizontal designs, and at the same time a simpler and more familiar form of program input. The reader may compare the example of microprogramming given in Weber (1967) with the program style of any of the machines mentioned above, which bears comparison with a conventional assembly program listing except for the primitive nature of the arithmetic, the absence of address modification, and the elaborate field selection and branching functions.

In moving to vertical coding it is normally the case that the main memory system has a much higher data rate than the host needs, even with the fastest control store. The extra capacity is used in direct memory access by I-O devices, in dual processor configurations, and in many instances by using the main memory as a source of microinstruction. The last option is particularly attractive because it affords an escape from the rigid limitation

on microprogram that is imposed by a separate control store. On the other hand it does impose a control structure which is difficult to rationalise: perhaps the simplest view is to look upon the interpreter as providing system standards, operating system interfaces, protection, etc, which are not normally present at the microcontrol level.

The following subsections correspond to the main design areas noted in the last lecture, with illustrations drawn from the machines mentioned above. Further examples can be found in less readily accessible specifications for many machines currently on the market.

## 2.1 Ceneralised Arithmetic and Data Paths

One of the obvious ways in which MSI or LSI components affect the arithmetic system is in allowing register lengths to be standardised at a reasonably high value, rather than making use of specialised lengths seen in earlier machines. The effects are to speed up the machine and to save control memory, because operations previously performed by a loop of microinstructions can now be carried out in one.

The host is still specialised with regard to arithmetic width and shift paths. Two methods have been employed for variable precision arithmetic up to a prescribed field size:

(i) using a third input to the ALU, which is in fact a mask allowing carries to propagate. The SCC MPL-900 allows the microinstruction to select one of 32 possible masks which can be used to propagate carry to the 'normal' sign position. A mask may also be used to permit operations on unpacked fields such as 6-bit characters stored in byte positions. One of the difficulties of working with unpacked data, however, is that it may eventually have to be aligned to an external interface such as the store address bus.

(ii) allow the effective ALU width to be variable, i.e. taking sign, carry and zero-test signals from any position of the ALU. This method is used in the El emulator and the B-1700, where the sign is part of preset control. If there are more than one arithmetic widths in use concurrently it is desirable to have more than one preset sign position, selected by microinstruction.

Variation in ALU width has an obvious counterpart in shift functions. To reproduce exactly the shift patterns of a word of arbitrary length it is necessary to preset the point at which end connections are made, which is more difficult to engineer than sign adjustment because a stream of bits is being handled. The El emulator does allow shift lengths from one to 64 bits, but the

logic is expensive and most designers have settled for single or double length shifts and rotations. For high level language interpretation that is probably sufficient.

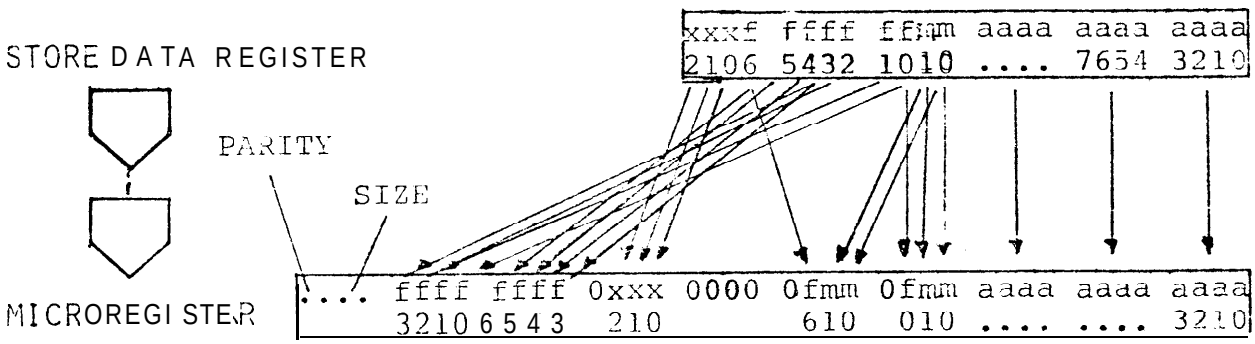
A final area where both the ALU and shifter are affected is in the type of arithmetic carried out. The predominant types are binary integer, decimal, and floating point. Generalised facilities for the last are usually complex and of limited value in either the commercial or research context. Decimal facilities can be built into the ALU in varying degrees, from fully signed operations down to facilities for detecting carries at the decimal digit positions. The choice rests entirely on the final cost/ performance required. Although an important area of design it can be 'factored out' in comparative studies of language-oriented and fixed instructions set machines, for which reason I shall not extend the discussion at this point. It is important to remember that if a host has good arithmetic facilities then any lapse in handling the control or data access side of a language will be conspicuous, and conversely.

If the path from memory is not selective enough (and it usually is not) facilities are required for extracting fields from micro-registers for input to the ALU. Such facilities are expensive and may be confined to limited field selection or to particular registers (e.g. in the shift unit). Thus, the B-1700 provides full extraction on one 24-bit register and 6-bit subfield addressing on most others. The El emulator can extract any byte from the 15 microregisters for comparison or control purposes. The MLP-900 can conveniently use the third ALU input to select fields within registers. Apart from the obvious hardware cost of selecting any field in any register, space will be taken to identify the field in microinstructions. It does not appear that high levellanguages demand complete generality, and limitations could be accepted simply on the grounds of coding efficiency.

## 2.2 Memory Mapping and Address Translation

The unstructured nature of machine codes, allowinginstructions to be used as data, and vice-versa, requires a strict correspondence to be maintained between the target machine and its representation in the host. (There are exceptions: in mapping the IBM 1401 onto the IBM 360 it is more convenient for the latter to use EBCDIC character codes, converting to and from BCD in those instructions sensitive to BCD formats). In most instances the target machine word is 'rounded up' when necessary to fit the host, not attempting to make use of every bit in store. However, the B-1700 goes to the length of resolving memory addresses to the bit level and allowing any string of up to 24 bits to be read or written, starting (or finishing) at a given position. In that case 100% memory utilisation can always be achieved.

The memory word or part-word is made available for analysis in the microregisters. It is an advantage to be able to select from two or three potential data registers in order to avoid extra 'move' microinstructions. At this point there is also the opportunity to map the data into a more easily managed form. The 'crosspoints' of the El emulator and 'language boards' of the MLP-900 both allow the choice by program of alternative hardwired data paths to and from memory. They may be used, for example, to prepare an instruction for decoding, to align 6-bit characters to 8-bit byte boundaries, or to handle parity conventions on a 'foreign' data bus. The diagram shows the cross point paths used by El to read ICL 1900 instructions, which enable function, register and modifier fields to be accessed without shifting the




target instruction microregister. The effect of the crosspointis to save 5 or 6 steps in the typical interpretive loop of 25-30 microinstructions. It can be seen as complementing the internal data selection functions: in a machine with powerful field selection orders crosspoints would be less important.

Apart from data, addresses have to be matched to the conventions of the host. For example, if the target machine uses decimal addressing and the host uses binary then conversion must take place before accessing the store. Similarly, if the target machine operates in virtual program space then virtual to real translation is called for. If page and segment table accesses are implicit in each memory reference the address conversion could easily exceed the combined steps of instruction decode and instruction execution. The alternative of using hardware assistance--allowing the host to work in virtual space--is expensive and still leads to delay in memory access. Fortunately, in the environment of high level language execution it is possible to work in a virtual address space but avoid most of the overhead of address translation.

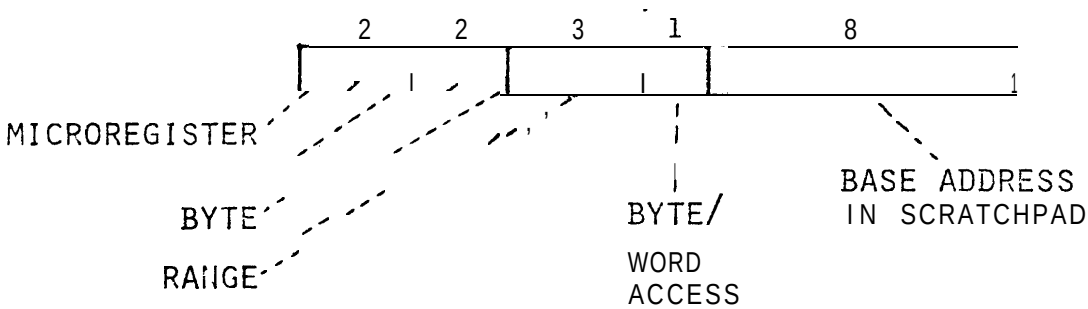
## 2.3 Representing the Target Machine State

The primary data of an interpretive program are the registers, the program counter, the instruction register, control flags,

channel status and control words of the target machine. A generalised host would expect to have room for the largest target machine state of interest, but even so it is unlikely to require more than a few hundred bytes of storage for that purpose, which often justifies a file of fast registers, the scratchpad (or local memory in IBM), in addition to the microregisters themselves.

It is a common requirement to access the scratchpad using an index value. For example, a target machine 'register-register' instruction contains two indices. Microinstructions do not admit the type of address calculation found in.machine instructionssets, therefore it is necessary to carry out some preliminary scratch-pad address calculation. That happens often enough--at least once in most target instructions--to justify building in predictive indexing hardware, which works in the following way. Certain microregister fields are designated (by preset parameters) as scratchpad indices. When any of those field values changes a scratchpad access is initiated (relative to a preset base), so that the corresponding scratchpad element is available for reading or writing in the next microinstruction (compare the main store address registers of the CDC 6600). The crosspoints for the El emulator are designed to place the target instruction

PRESET INDEX DESCRIPTOR




register and modifier digits in the position of predictiveindices, allowing the register and modifier values to be used without delay.

The primary data of a high level language machine are the intermediate results, control flags, and the control, stack and environmental pointers that allow access to contextually relevant data. For the most widely used languages the 'state' can be mapped into a register file quite easily; moreover, its access patterns correspond closely to those of conventional target machines, hence the scratchpad organisation of a 'universal emulator' is equally applicable to the major programming languages. Whether there are alternative organisations suited to a wider class of languages is a question we shall consider later: it might be argued that a language is 'major' because it happens to fit onto conventional hardware, and that when that constraint is removed more attention can be given to problem-oriented languages.

## 2.4 Generalised Control of Peripherals

At this point we must draw a broad distinction between emulation of the non-privileged users' instruction set and that of the operating system. The latter would include instructions for channel selection, requesting device status and sending commands as well as receiving and sending data. It may also include special addressing modes for channel control words, page and segment table control, interrupt register and timer access, handkeys, displays, fault indicators and so on. Full-scale emulation, to the extent of running the target machine's peripherals, engineering test programs, channel commands and operating systems involves at least twice the design effort of the non-privileged instruction set alone and will almost certainly involve physical. adaptation of the peripheral interfaces.

In the present context, recognising that most languages are non-specific with regard to the means of peripheral control, the preferred approach is to match the I-O statements to the host system using machine language and microcode procedures.

## 2.5 The Effect of Large Scale Integration

The level of complexity achievable inbipolar LSI devices has reached the point of presenting complete slices (2 or 4 bits) of control or arithmetic circuitry in a single package. However, such circuits are only realised in favourable commercial/technical situations, i.e. wide applicability and high functional content in relation to edge connection. Some of the machine features discussed above would fail on both counts. On the other hand, I have indicated that language execution makes less stringent demands then universal emulation, hence the 'generality' aimed at by device manufacturers may well provide effective support for the target instruction sets of interest in the context of high level languages.

**How** much does generality cost in terms of performance? That is impossible to say without detailed analysis of a range of target machines. An indication can be given by comparing the vertical encoding of the ICL register-store 'ORX' instruction on the El emulator with the horizontal form for the 1904E. In terms of microorders, the El obeys 30 compared with 14 for the specialised host. The difference is by sequence control (13:6), function decode (5:2) and operand access (10:5). However, the most startling figure in each case is the ratio of support activity to 'useful' function: about 15:l. Our main concern in designing language-oriented target machines must be to reduce that ratio.

## 3. INTERPRETATION OF HIGH LEVEL LANGUAGES

The existence of readily microprogrammed host machines naturally gives rise to speculation about the likely return from bypassing the normal instruction set. To do so succeefully involves the solution of a range of problems concerning definition, security, expansion, maintainability and so on, whose solution is taken for granted in conventional systems. Before looking at the broader problems it would be reassuring to have some measure of the potential advantage of microcoding, which is the subject of thislecture.

It is easy to find performance improvements in the region of 10:1 or more for a particular algorithm expressed in microcode compared withmachine code. In evaluating such figures it must be remembered that they derive from three contributing sources: (i) the inherent speed of microcode which is the result of the simplicity of the instructions and the use of high speed control Store; (ii) occasional advantages of the microfunctions over the target machine functions, especially in bit manipulation and control sequencing; and (iii) advantages gained from bypassing the architectural framework of the target machine, especially its protection mechanisms.

It would be meaningless to draw conclusions from isolated algorithms. The minimum basis of comparison is taken to be the combination of hardware and software supporting one of the major programming languages, which provides the syntax and semantics for a broad class of problems. The main parameters of performance are taken to be:

(i) compile and load time

(ii) execution time

(iii) size of the support system

(iv) object program size

(v) diagnostic aids in (i) and (ii)

The two techniques used for performance comparison are benchmark testing, in which space and time measures are obtained for a representative sample of source programs, and factoring, in which performance is inferred from independent measures on artificially chosen statements. From the design point of view the second is much more useful, though except in the case of Algol 60 there do not appear to be any widely published sets of reference statements. Needless to say, the object of design is to optimise performance at a given system cost over a prescribed set of languages.

The weights attached to the measured parameters will vary from one class of use to another and no attempt will be made to determine them here. The aim is to show how variations in processor

function--specifically those brought about by microprogramming--affect the parameters (i) - (iv). At the same time the qualitative effect of diagnostic aids will be assessed. It will be seen that the time measures depend partly on performance of a second language which will be referred to as the system implementation language (SIL), so whether the machine is good at compiling Fortran, say, depends on what it has to do to produce executable code, and how well it does it: as far as possible the second factor will be isolated by measuring the overall performance of run time support modules. Whichapplies also to execution of the functions of the language by stored microprogram or hardware because that does not usually vary from one language implementation to another and it can be measured in basic arithmetic speeds. It would be relevent, however, if one implementation chose to use a decimal radix, while another implementation of the same language on the same machine used binary. Most of the language implementations reported in the literature have been rendered useless from the design point of view by not keeping the executive algorithms constant: in other words, if a performance gain P is generated it is impossible to tell how much of P derived from the interpretive technique and how much from improved arithmetic or run-time support.

The following subsections make a broad distinction between procedure coding, illustrated 'by some of the scientific languages, and data access, which is examined in the context provided by Cobol.

### 3.1 Algol, Euler and Expression Evaluation

Factored measurements of Algol performance are reported by Wichman (1973). In Table 1 I have abstracted some figures for machines with roughly comparable arithmetic times. It is well known that the Burroughts B-6700 uses a target instruction set tailored to the representation of Algol: its effect can be seen in the times for procedure entry. One would also expect it to be effective in array assignement, but in this particular case the compilers spot the indices [1,1] etc and generate optimised code for the conventional machines. The advantage of the language-oriented code is to simplify the compiler rather than speed up execution.

The importance of individual statement times depends on the weights attached to them in the final performance measure. In general, arithmetic and array access operations have the highest weights, procedure entry is an order of magnitude less important, and array declarations an order of magnitude less than that. It must be remembered that experimentally observed times reflect a complex combination of hardware, software and support system. Implicit in many decisions is the designers' assessment of different language features, and his budget reflects an assessment

of the importance of the language as a whole.

TABLE 1: SOME ALGOL STATEMENT EXECUTION TIMES

| Statement | Execution time in microseconds | | |
|---|---|---|---|
| | B-6700 | IBM 370/165 | Univac 1108 |
| x := 1.0 | 5.5 | 1.4 | 1.5 |
| x := 1 | 2.7 | 1.9 | 1.5 |
| x := Y | 3.9 | 1.4 | 1.5 |
| X := y + z | 5.5 | 1.4 | 3.4 |
| X := y * z | 11.3 | 1.4 | 4.0 |
| e1[1] := 1 | 5.3 | 1.6 | 2.7 |
| e2[1,1] := 1 | 7.7 | 1.7 | 5.8 |
| e3[1,1,1] := 1 | 11.3 | 1.7 | 9.0 |
| begin array a[1:500];end | 408. | 242. | 918. |
| p1(x) | 28.6 | 60.7 | 127. |
| p2(x,y) | 30.5 | 83.6 | 137. |

[Note: The times for the IBM 370 probably err on the low side because of the effect of the cache]

In comparing object code size, Wichman gives the following figures normalised with respect to Atlas:

| | |
|---|---|
| Burroughs B-5500 | 0.16 |
| Univac 1108 | 0.31 |
| CDC-6600 | 0.56 |

The advantage of the Algol-oriented intermediate form in comparison with some of the best conventional systems is evident. To understand how such results are obtained we must examine some target machine states and the functions applied to them.

The advantage of language-oriented intermediate code is that, provided an 'expression-evaluation' mechanism is built in to the interpreter, the details of register transfers that are usually found in machine code can be omitted. The compiler is simplified, the code is more compact. It is not inherently faster, because the data access is indirect, but in many instances that is more than compensated by savings in other parts of microprogram. The stack mechanism is the best known means of expression evaluation: the reader is no doubt familiar with the reverse polish form of code used in Burroughts B6700 and other mechines and the various stack and environmental (display) pointers associated with it.

However, the apparent simplicity of the Burroughts representation leads to some complexity in the machine functions themselves. The value call operator (VALC) has to be able to detect and interpret all the operand types that can legitimately be presented in the course of computation, including indirect references through the stack and procedural definitions arising in parameter lists. In most applications the questions answered by examining tags could be answered in advance by the compiler: as a general rule unnecessary tests at execution time should be avoided except as deliberate backup for the compiler, the support system or data security.

In contrast, dynamic tag testing is essential to languages such as Euler and APL because the type of a variable is not predictable at compile time. Let us examine the Euler representation in greater detail and see how one of the target machine syllables fits onto the architecture of the IBM 360/Model 30 described in the first lecture (for greater detail, see Weber (1967)).

The representation of a variable is a [tag,value] pair, the tags having the following significance:

| | | | |
|---|---|---|---|
| 0 | Null | 5 | Reference (m,loc) |
| 1 | Integer | 6 | Procedure (m, link) |
| 2 | Real | 7 | List (length, loc) |
| 3 | Boolean | 8 | (Unassigned) |
| 4 | Label (mp, pa) | 9 | Block mark (in stack) |

The run-time environment consists of three storage areas: Program, which is indexed by pa (program address) and link(returnaddress); Variable, indexed by loc (location), where all defined data is to be found, and the Stack, which consists simply of block marks giving static and dynamic chain links, references to parameters in the Variable space, and intermediate results. Operators exist to test the tag of a variable, e.g.

isn A  Is A an integer?

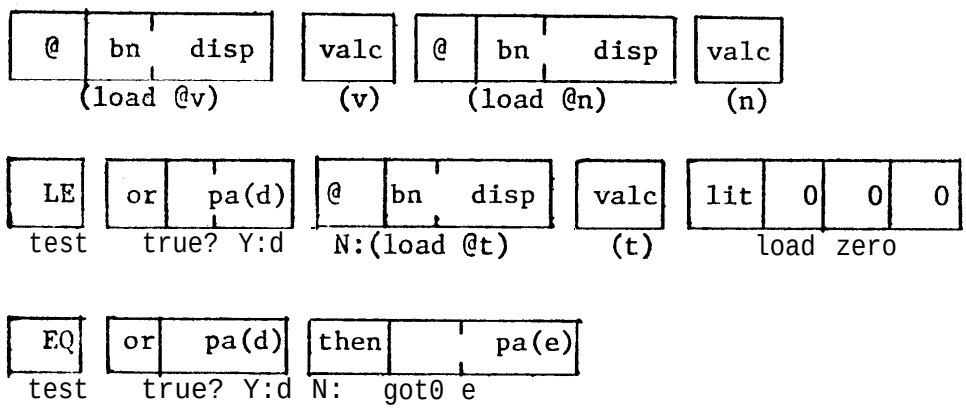
returns the boolean value true or false. Standard operators such as + - * / mod max abs can be applied to numeric values, yielding numeric results, and failing if illegal tags are encountered.

A list is an ordered set of values, each of which is either an elementary type or a list. Lists can be created dynamically, and operators exist for enquiring the length, detaching the tail, selecting an element and concatenating two lists. The existence of reference variables causes the variable space to be maintained by scanning pointers and recovering space which is no longer referenced, updating pointers when compacting the active store areas.

The Euler program area consists of sequences of operator syllables (bytes), each followed by the appropriate number of bytes giving literal values or indices. The program is represented in reverse Polish form, e.g. the statement:

'if v≤n or t = 0 then d else e'

would be represented by the following string of 27 bytes:




Note that the @ operator forms a reference on the stack, which valc converts to the corresponding value. The translation is thus a simple reordering of the input string, replacing variables by [block number, displacement] pairs. The latter are converted into [mark number, loc] pairs on loading to the stack. In the program the logical connectives give a destination to which control passes if the top of stack element has the required value. Figure 4 gives the microcode for the and, or and then operators. A Boolean variable has the binary form '0011000y', i.e. tag 3 and value y = 1 for true. The microregisters IJ are used as program counter, UV points to the top of stack. For simplicity, the address incrementing microorders, which are really byte-serial, have been written as 'IJ + 1' etc.

The sample microsequence checks the tag of the operand and interprets the logical connective in 8 microinstructions, 4 main memory cycles, or 6 µsec (7.5 if false). The corresponding IBM 360 target instructions would take the form:

```
CL1     O(STACK), LOGT
BE      ORTRUE
CLI     O(STACK), LOGF
BNE     TYPERROR
SH      STACK, ='4'
```

The interpretation of that sequence takes 32 µsec if 'true', 90 µsec if 'false'. It occupies 24 bytes of program as opposed to 3. That puts microprogram interpretation in its most favorable light: dynamic type assignment, minimal arithmetic content and naive compiling techniques. It is easy to see that even with dynamic type -assignment it is often possible for the compiler to

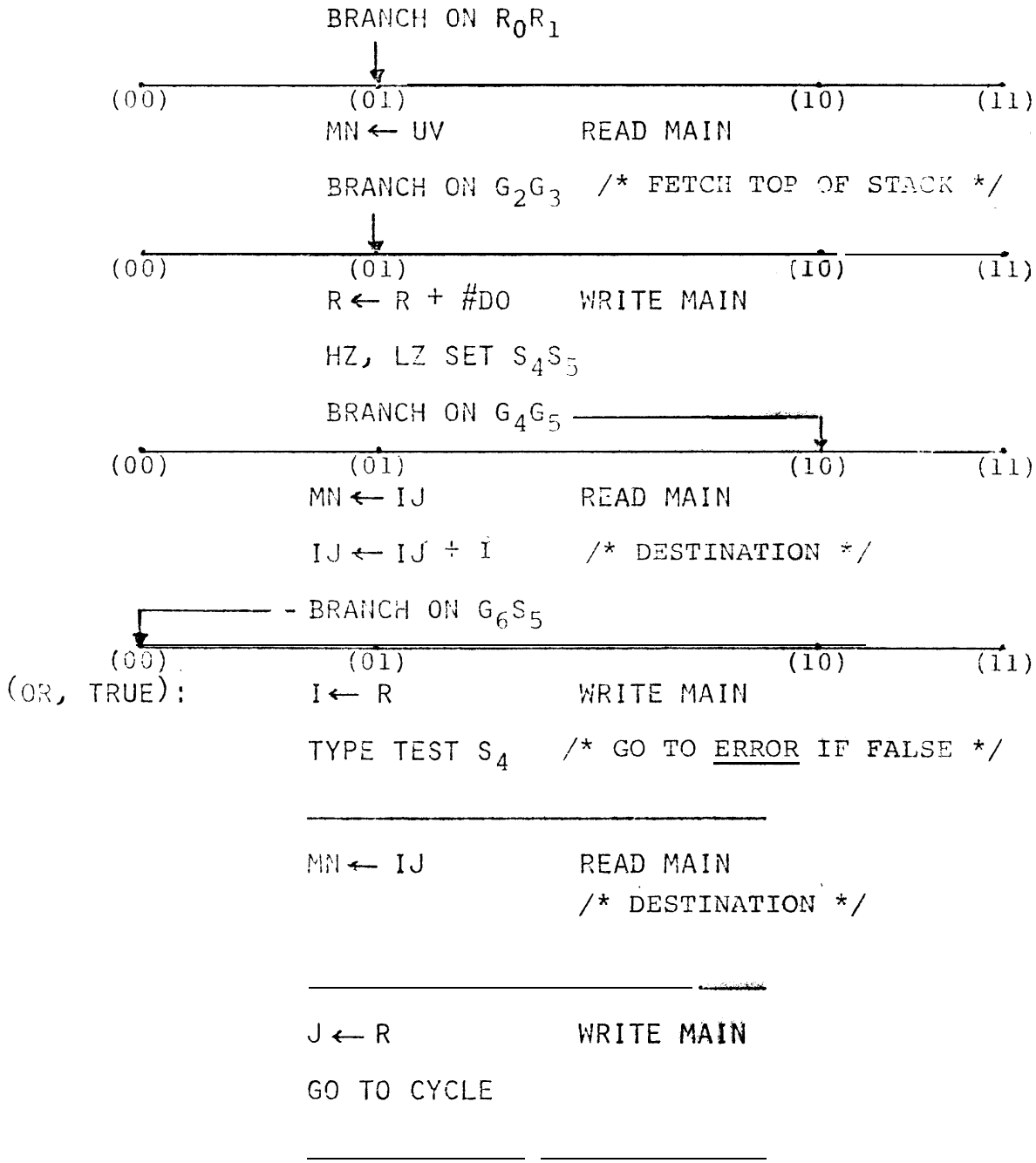



Figure 4: Microcode for Euler Logical Connectives

predict the result of an operation as far as type is concerned, and to omit further checks, as in:

```
if x = y ....
```

which must give a Boolean on top of the stack.

The advantage in space which results from the syllabic form of target instruction is a combination of two effects: the localisation of the operator/operand space implied by the source language, and the use of working registers implied by the stack. It would be possible to compress an operand'address' to 3 or 4 bits, for example, provided changes of 'context', in which the full meaning of the operand is expanded, can be effected without excessive overhead. Unfortunately, very little is known about the consequences of one choice or another; it is not even clear that procedure boundaries should play a part in defining context. The use of a stack mechansim may not be optimal: we can see that some run-time maintenance activity is involved of which a compiler could avoid, and it is known that the majority of expressions found in practice are of very simple forms which do not require the full generality of stack evaluation. Hoevel and Flynn (1977) suggest an alternative primitive form of instruction which recognises many important special cases. Space gains of up to 5:1 for Fortran compared with IBM System 370 optimising compiler are reported.

### 3.2 Cobol Interpretation

The major parts of a Cobol program are the Data and Procedure Divisions. The program operates on files of records and uses internal records for workspace. Each possible record format is declared in the Data Division: the same physical record may be mapped according to many different declarations, so there is no question of concealing representations or placing descriptive tags as parts of the record. The elementary items of data have a wide variety of representations with a dozen or so basic data types. The elementary items are named, and may be collected into named groups, which in turn may be grouped, up to the level of the record name itself. With the aid of PICTURE descriptions editing characters can be inserted in a field for output (and conversely for input) with the result that the 'type' code associated with a data item can be of almost any length.

Within a record individual items or groups of items may be repeated. The number of actual occurrences may vary, depending on a field in a fixed position in the same record. Repeated items are selected by following the repeated group or field name in the Procedure Division by one or more subscripts, or by using an implied Index value. The coefficients of the associated storage mapping function can be determined by the compiler.

The Procedure Division is composed of a number of Segments, whose significance derives from the days of programmed overlays. A Segment comprises a number of labelled paragraphs, each containing one or more sentences. A sentence consists of one or more Cobol statements.

Individual statements have a fairly simple syntax, a verb followed by data names and Segment or paragraph names, e.g.

ADD P TO Q GIVING DAY-TOTAL ROUNDED

where P, Q and DAY TOTAL are data names. The definition of Cobol implies strict observation of decimal rounding and truncation and is subject to the types of operands and the size of intermediate results (18 digits). The compiler is required to indicate if operands are incompatible. or if intermediate results are out or range. Some indication of verb frequencies is given by the following measures from a benchmark test:

| VERB | DYNAMIC USAGE | STATIC USAGE |
|---|---|---|
| MOVE | 30% | 33% |
| TF | 30% | 18% |
| GOTO | 11% | 19% |
| ADD | 10% | 6% |
| PERFORM | 7% | 8% |
| WRITE | 4% | 3% |
| READ | 3% | 2% |
| Others | 5% | 11% |

Thus for execution purposes seven verbs account for 95% ofexecuted statements, while the same seven account for almost 90% of stored statements. The target code can be chosen purely as a compromise between compiler and microcode, without concern for reconstructing the source string (which affects APL coding for example). The final form depends on what are regarded as reasonable limits for field sizes in one Cobol source module. In the target instruction listed in Table 2 the maxima are taken to be:

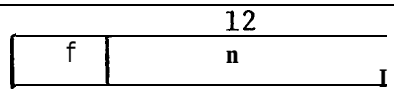
Variables: 4096 ; Indices: 256 ; Files: 256 ; Data areas: 64

Procedure variables: 256.

In the design used here, which is based on a Cobol interpreter written for the ICL E1 emulator, each Cobol statement is represented by a sequence of 16-bit target instructions.

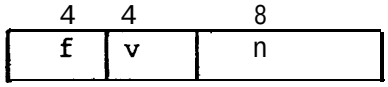
CABLE 2: A COBOL TARGET INSTRUCTION LANGUAGE

Format #1




f=0: Source operand at DQT[n]
f=1: Destination at DQT[n]
f=2: Operand at DQT[n]
f=3: Operand n
f=6: Branch within code area, offset n

Format #2




f=7: n-byte literal operand, type v
f=8: Scale operand, partial result,..., by n
f=9: Arithmetic; scale first operand by n
v[ADD, SUBTRACT, SUBTRACT-GIVING, MULTIPLY
DIVIDE, DIVIDE-REMAINDER, ..., etc]
f=10: Branch DEPENDING, via Procedure variable n
f=11: Branchn, depending on condition v
f=13: v[MOVE, COMPARE, SET INDEX, DEBUG, STOP,
and call RUNTIME support]

RUNTIME: ACCEPT TIME, DATE, DAY, DISPLAY,
OPEN, CLOSE, READ, WRITE, REWRITE, START, DELETE,
CANCEL, CALL, EXIT, etc.

Cobol control structure is the source of some complexity because of the use of procedure variables and debugging options. Apart from the normal branching determined by GOT0 statements it is possible to specify that a particular paragraph or sequence of paragraphs should be PERFORMed one or more times, or until a condition is satisfied (possibly varying some elements on each repetition). A simple compiler cannot tell in advance which paragraphs will be the subject of PERFORM, so it will insert a possible branch to a 'procedure variable' at the end of each paragraph: if PERFORM does not apply, the branch 'drops through' to the next paragraph in sequence. Further complication derives from the ALTER verb, which can be used to change the destination of a GOTO. Rather than change the stored object code the branch is again directed through the procedure variable table.

The complication arising from debugging is that any attempt to access a named data item, paragraph, file or index may be required to enter a debug procedure. In most compilers that means that the code generated for handling debugged elements is different from (and slower than) normal code, even when executing with DEBUG OFF. In interpretive systems the same target code is generated in all cases and the branch is taken in the interpreter.

In the Data Division all names are mapped unambiguously into indices in the lists of data qualifiers (DQT), file and index table. Procedure variables are indexed in the Procedure Division. Information built up during the compilation phase can be carried over into execution without change in many cases. Figure 5 shows the modular structure of Cobol as far as it affects the interpreter. The DQT contains a 64-bit descriptor for each variable, giving:

- the index of the base pointer for the record currently containing the variable
- offset and limit of the variable within the record area
- whether the debug option applies
- operand type and scaling information
- it subscripted, the index of mapping parameters in the subscript information table
- if edited, the index of editing parameters in the edit information table.

At runtime the data qualifier element DQT[n] is interpreted to give the address pointer to a sequence of bytes (or bits) within the area defined by the base. About 20 microsteps are required to extract the data attributes and place them in microregisters, followed by whatever is needed to extract the data itself and present it for the next operation. Hence the management of the DQT represents a significant part of the interpretive overhead.

In measuring Cobol performance the time and space requirements of a set of test statements were measured, and final figures of merit obtained by weighting the results according to dynamic or static usage. For space, a gain of 1:3 resulted in comparison with the ICL 1900 program requirements. It appeared possible to improve on that by adding to the function set. For time, an overall improvement of 1:2.5 was observed in comparison with the conventional compiler on the ICL 1900. That figure is disappointing. It is accounted for in part by the arithmetic complexity of Cobol. Nevertheless the average Cobol statement appears to need about 200 microsteps (as opposed to 500), and in several instances the conventional compiler generates code that runs faster than the interpreter, for much the same reason as we saw earlier in looking at Algol implementations. However, another factor proves to be significant: the time spent in the interface between the language interpreter and the supporting SIL.

## 4. INTERPRETIVE SYSTEM DESIGN

Improving on the range-defined instruction sets of fifteen years ago without meeting comparable system objectives is not particularly difficult. To present a realistic alternative it must be shown how programming standards can be maintained through a very wide power range; it must be 'possible to develop and maintain new languages and subsystems taking full advantage of the architecture without endangering system security; storage and control structures must be created to suit modern applications rather than those of the early 1960's. As far as I know, no 'microsystem' has been developed with the required properties. Even so, it is not sufficient to show that variable microcode achieves better results than fixed instruction sets: we also need to be convinced that it is the best way of using modern technology. In this lecture I shall draw together some of the results observed in language-oriented machine design and suggest two alternative system frameworks in which the demonstrated advantages could be retained.

### 4.1. The Effect on Language Parameters

As I have already indicated, many of the measures of language performance are affected strongly by the choice of supporting system, which we suppose to be reflected in the semantics of the System Implementation Language (SIL). For example, suppose the SIL is in fact a copy of the Executive package of a conventional machine range, and that a Cobol application package is obeyed (a) using the fixed instruction set and (b) using a Cobol target code such as discussed in the last lecture. Then the observable effect on storage requirements would be as follows (using typical figures for the ICL 1900):

| | (a) Fixed Instr. | (b) Fixed+Cobol |
|---|---|---|
| Fixed instr. µcode | 16 Kbyte | 16 Kbyte |
| Cobol target µcode | 0 | 9 Kbyte |
| Executive (kernel) functions: | 16 Kbyte | 16 Kbyte |
| System functions (spooling, command language, etc) | 20 Kbyte | 20 Kbyte |
| Cobol run-time support: | 25 Kbyte | 25 Kbyte |
| Cobol application - data (say) | 9 Kbyte | 9 Kbyte |
| - code (say) | 9 Kbyte | 3 Kbyte |
| Total | 95 Kbyte | 98 Kbyte |

In other words, the reward for a great deal of effort and investment in control memory is negligible as far as storage is concerned. Of course, one can present the picture in other ways and use the speed gain to advantage if there is sufficient I-O capacity, but the point remains that unless the support system gains similar advantages from the interpretive techniques the improvement in language performance will be seriously diluted. Let us assume,

therefore, that the SIL itself benefits from the use of micro-program. The effect may be seen as space reduction and a gain in speed; more probably it will be seen as improvement in function and flexibility. In reviewing the parameters listed earlier some of the requirements of the SIT., will be noted.

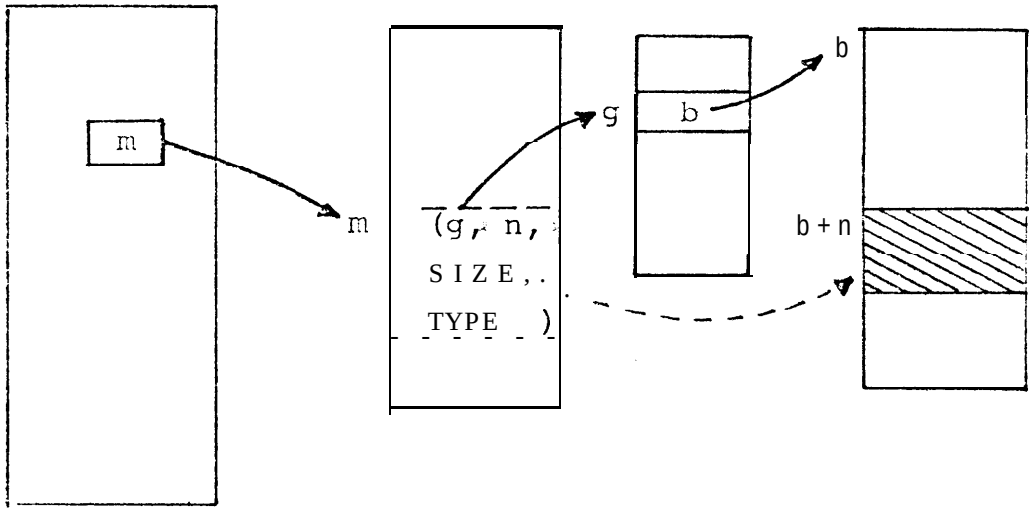
(i) Compile and Load Time.

Substantial (say a factor of 5) gains in speed can be made in the portions of a compiler concerned with lexical and syntax analysis, and to a lesser extent in code generation, by microcode interpretation of syntax tables. Where in-line coding has been used in the past the speed gain is smaller but significant saving in space is achieved by table-driven techniques. Compile time is indirectly affected by the choice of object code under (ii).

Load time is normally determined by the supporting system. If all programs have to be mapped into a (virtual or real) linear store the time and space overheads in starting a job step may be significant (comparable with the compiler itself in many conventional systems). Moreover, the operating inconvenience is significant and may result in such anomalies as separate 'batch' and 'load-and-go' language systems. There is no reason, however, why the SIL functions should not allow program execution with explicit structure. For example, the operating environment shown in Figure 5 can be maintained with no appreciable execution overhead on the part of the SIL. In that case, the load time is negligible.

(ii) Execution Time

Excluding arithmetic and I-O, execution time is governed by the time of access to variables and the change of control environments, i.e. the subsets of the program space immediately available from particular points in the program. It is the 'localisation' of the environment which allows short addresses to be used and produces the greatest contribution to code compaction. The diagram shows the components of a generalised access chain. Data elements are assumed to be created in blocks (activation records or file areas) which are not necessarily contiguous in store, but selectable by an index n. Data identifiers in the source text are mapped into indices m, which are used to refer to a table of attributes (cf the DQT in Cobol) which give record pointer, offset, size, type, and possibly other information derived by the compiler and required during execution. In general, several sets of attributes may refer to the same record, and one set of attributes can refer to several record areas (through dynamic adjustment of the control environment).




OBJECT CODE ATTRIBUTES CONTROL ENVIRONMENT DATA STORAGE

(STATIC . . . . . DYNAMIC)

Languages differ in the amount of attribute information carried into the execution phase, the method of changing the control environment, the time at which attributes are assigned, and hence in the ways of distributing components of the access chain in storage. In Fortran, for example, attributes and record pointers can be absorbed into the object code; in APL the object code and attributes are dynamically assigned; in Algol the (g,n) pair and size can be absorbed into the object code while the type is sometimes attached to the data in the form of a tag. Where explicit maintenance of attribute and environment is demanded by the language there can be significant gains from using microcode. The ratio of addressing and control instructions to arithmetic in the output of a conventional compiler is in the region of 4:1, so assuming a 5:1 speed increase from microcoding the former an overall speed gain of 5:1.8 or 2.8:1 is indicated. One would expect more for the highly structured or 'dynamic' languages. Further speed gains can be expected where specialised arithmetic functions are called for, e.g. array, complex, controlled precision or character string manipulation. A minimum overall gain of 3:1 in speed of a 'production' compiler to range standards would be a realistic objective for the languages in common use.

A language allowing free assignment of pointers (reference variables) entails potentially serious support overheads in the assignment and recovery of space, not necessarily eliminated by the provision of a large virtual store. Even if the SIL recognises pointers it seems preferable for the language subsystem to undertake its own space management to take advantage of known local characteristics. The language 'pointer' is evaluated in terms of the underlying program structure at the time of use: that operation occurs frequently and benefits from processor adaptation to the extent that once an evaluation has been carried out the result can be used repeatedly on successive items of data. It is then required of the SIL to allow language interpreters to work with

'absolute' as well as virtual addresses. In the next subsection we shall see what that implies. (The alternative of having both the SIL and the language microcode work in a virtual space supported by hardware can be disregarded because of the delay in accessing memory and the poor store utilization that results.)

Space management functions are principally concerned with searching for and updating pointers and physically moving blocks of data. They are time consuming and in many languages their use is discouraged by artificial means, so the gain from making them more efficient would be seen in program flexibility (in the user language and the SIL) rather than in execution time.

(iii) Size of Support System

The SIL code benefits in two ways: in many situations, e.g. in compiling to language-oriented code, it has to do less; and it does it more efficiently than other high level system programming languages, or more elegantly than a macroassembler. Size reductions in the region of 5:1 have been achieved for compilers. Each language microcode represents a space overhead of at least 10 Kbytes, plus a similar amount for the resident SIL.

(iv) Object Program Size

Tailoring the object code to fit the source language shows the clearest gains over conventional systems because of the elimination of unnecessary function, register and address bits. An overall reduction in procedure size of 4:1 for large programs, including attribute tables, would be a realistic aim. No significant gains in data mapping over a conventional system with word and character addressing can be expected. Gains in space can be seen as gains in main memory and channel capacity and to a smaller extent in file space.

(v) Diagnositc Aids.

As any APL user discovers, interpretive methods can give exceptionally good diagnostic information, sufficient to overcome eccentricities of the language itself. Unfortunately, diagnostic quality is one that cannot be measured and is often overlooked in favour of marginal improvements in the others.

## 4.2 Microsystem Problems

The use of microprogram brings its own problems, and raises the question of whether the implied comparison with machines of the mid-60's was the correct one to use. In the system context, the obstacles to using interpretive microprogram are as follows.

(A) Kange Definition

The microprogram appropriate to a high performance machine is quite different from that of a slower microprocessor. There is also an absolute speed limitation: a machine executing target instructions at 10 MIPS is obeying microorders at least 10 times as fast, which is beyond the power of vertically encoded (i.e. easily programmed) host machines.

(B) Security

Microprogram derives part of its speed advantage by ignoring the security checks inherent in fixed instruction sets. For a small amount of microprogram under control of the manufacturer that is tolerable. The language performance figures obtained in practice give the interpreter responsibility for resources normally regarded as protected, i.e. absolute addresses, in which case the security of the system is in the hands of language implementors.

(C) Flexibility

Microprogram is a static form of code. It cannot easily be moved in store. Fast control memories and scratchpads are necessarily small, so the problems of sharing resources between interpreters and scheduling their use have to be solved.

Of the above, (B) alone is sufficient to prevent widespread use of microprogram in commercial systems. Four types of response can be recognised:

1. Embed the Microprogram in a Conventional System

We have already noted that the space and time advantages are diluted in the context of a conventional system, nevertheless, those that remain are obtained with minimum investment in redesign. The IBM APL Assist Feature running under DOS/VS, OS/VS1 and OS/VS2 has been made available on the System/370 Models 135, 138, 145 and 148 (Hassitt and Lyon (1976)). It consists of an additional 20 Kbytes of microprogram, resident in main store, which interprets APL statements. It carries out virtual--real address translation according to the rules of the host system, but returns control to the host to service interrupts and page faults. Hence,system integrity depends upon correct use of addresses in the APL microcode.

2. Extend Security Boundaries to the Microprogram Level

The in-line checks that can be used without impairing performance are restricted to key comparison, lockout on fixed sized

blocks of store, etc. The El emulator provides write protection on 16-word frames of scratchpad, 64-word frames of control memory, 16 kword Frames of main memory and all I-O multiplex positions. The main drawback to such schemes is their inaccuracy and the difficulty encountered in handling dynamically changing or moving programs, which occur quite frequently in modern systems.

3. Control Address Formation in Microcode

An alternative, which can be seen as a generalisation of the first approach, is to validate addresses when they are formed, then to restrict their use so that further checks are unnecessary. The SIL is responsible for forming addresses (from segment capabilities); the language microcode can modify them within given limits and access the store directly. Addresses are distinguished by tags so that the SIL can find and update them when necessary, independent of the source language. This method is used in the Variable Computer System (Iliffe and May (1974)) on the El emulator, which makes provision for tag manipulation. For complete security, however, specialised hardware support is necessary.

4. Separate the Language Processors Physically

A special case of the second approach, which is attractive because technology is available in the form of low-cost microprogrammable machines. The separation is conceptually physical, in the form of multiple processor-memory pairs, but it could be achieved by time-slicing.

From the general design viewpoint either of the last two approaches can be used to provide a viable system model. Each intends to cover a wide range of performance by using multiple computers. From 3 it can be seen that because access to program space is controlled the SIL and user programs can coexist in the main memory and control store (if it exists), and that programs can be distributed over the available memory space. This 'distributed program' model is well suited to the class of applications with dynamically changing program requirements, or which can be expressed in terms of cooperating parallel processes.

From 4 a more specialised 'dedicated language' model isderived. Each program, together with its interpreter, has unrestricted use of the local memory space of a processor-memory pair during execution, but it is rolled in and out by the scheduler whichforms part of the SIL. The STL microcode and system procedures can be protected by holding them in read-only memory. Access to shared data or to overlays must be through some form of secondary store manager, which checks the rights of the user against declared accessibility of the data, a relatively slow operation. The disadvantages of the dedicated-language model are the sensitivity of programs to physical store sizes, the amount of unproductive

traffic between central (i.e. secondary) memory and language processors, the poor utilization of processor and memory resources (if it is argued that processors and memory are give-away items, why bother with microprogram at all?). Nevertheless, such a system is in many ways the easiest to understand, it is least affected by failure of one of the processor-memory pairs, and it lends itself to the 'personal computer' mode of working in the same way that private cars lend themselves to private transport, however inefficient.

Each model presupposes the use of a system implementation language (SIL) whose aim is to provide a set of functions that can be used in all language applications to reduce development effort and code duplication at both micro- and target machine levels. In so doing it sets standards that can also be used in the variable part. There is no doubt that certain operations such as input-output and frequently used arithmetic procedures are properly part of the SIL. How far one can go depends on the type of system: if the integrity of system data cannot be guaranteed (which is the case for dedicated-language models) the amount of support the SIL can give is limited. On the other hand, commitment of the SIL to support facilities that are rarely used complicates the system and wastes resources. The interesting design area is thus the 'fringe' of functions just inside or just outside the SIL, which I can best illustrate by reference to the Variable Computer System developed on the El research emulator and later transferred to another host machine.

### 4.3 An Example of a SIL: The Variable Computer System

VCS is implemented at two levels of control: microprogram and the system target language (VCSL) in which all compilers and system utilities are written. The VCS procedures can be called either at microcode or at machine code level. It follows that if a microprogrammed procedure is called from machine level, or vice-versa, some code must be obeyed to adapt from one level to the other. It is undesirable to impose restrictions at this point because one cannot always predict whether a procedure will be committed to microprogram; the descrimination must be dynamic or immediately before task initiation, at worst. For that reason the list of procedure activations associated with any process contains both micro and machine level linkage information. Again, it is undesirable to impose limits on the depth of procedure call, therefore linkage information is stacked in main memory, the host machine link stack having very limited use.

Procedure activations form part of the process state vector (PSV), which also contains VCS registers, environment pointer, current program pointer and various flag bits that are mapped into the host registers. As calculation proceeds it is possible that other host registers will be used, but it is required that all

state information will be contained in the PSV at points where a change of procedure or process may occur, In that way the VCS can effect process management without explicit knowledge of the language state, arid with a fair degree of independence of the host machine. Similarly, by recognising tagged addresses the VCS can carry out store management without explicit declaration of the mapping used in current processes.

Procedure entry and exit is controlled through a dynamic chain of marked links. The purpose of the marks is to distinguish task initiation, system call and user procedure calls, allowing various levels of restart to be employed and providing excellent diagnostics at both control levels.

The interpretation to be placed on a program segment is indicated by a control type assigned to a particular compiler. Control type zero is used for pure data: any attempt to obey it will fail. Control type 1 is for system use, type 2 for VCSL target code, and type values for language extensions, e.g. to Cobol, APL, etc, are assigned 3, 4, . . . on a global basis. The control type is examined on procedure call and return (in the case of machine level code), branching to the appropriate interpreter.

It can bee seen that the PSV's are key control structures that must be protected if system security is to be ensured. The most efficient and flexible basis for protection is a capability scheme such as that of the Basic Language Machine. Many of the VCS functions are concerned with creating and manipulating abstract system objects in a consistent way, the PSV's being the representation of the abstract idea of a 'process'. In particular, we find functions for:

(i) setting up operating environments (bases) and defining the resources found in them;

(ii) creating, starting and stopping processes;

(iii) entering and leaving procedures;

and (iv) controlling access to resources.

Here a 'resource' is a storage segment, PSV, I-O device, or a set of resources. The recursive nature of this definition allows each base to be constructed as a tree. Clearly, the integrity of any object depends in the end on maintaining the integrity of its representation, i.e. the store, and of the procedures that are applied to it, i.e. the activation records contained in the PSV's.

Program structure is dynamic. A new base is able to share the information available to its 'parent' at the time of its creation, with the effect that a hierarchy of bases is set up with the 'system' at the apex. The base structure is important in building

language subsystems and dependent application environments: Figure 5 shows a typical three-level base structure to which one or more Cobol modules might be attached.

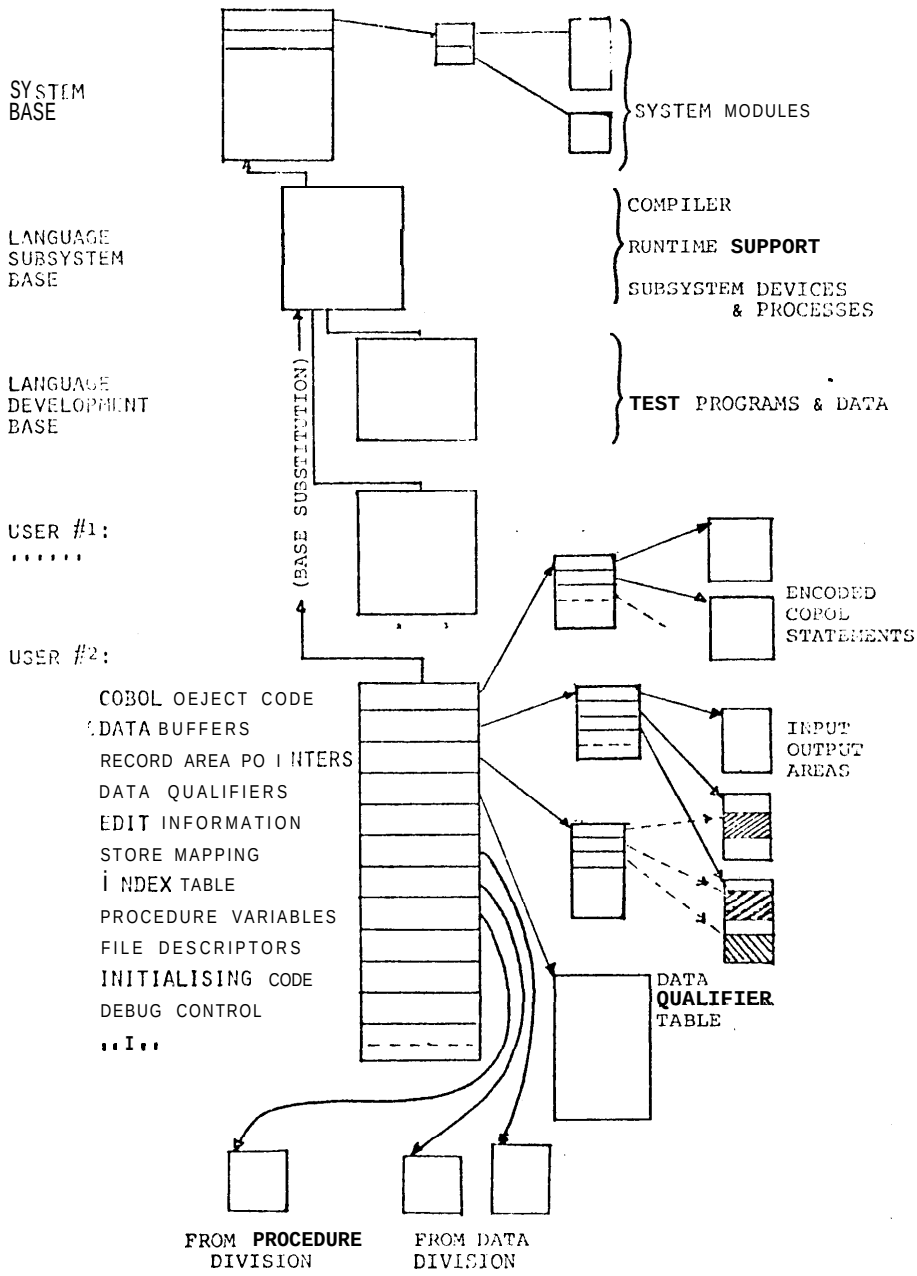
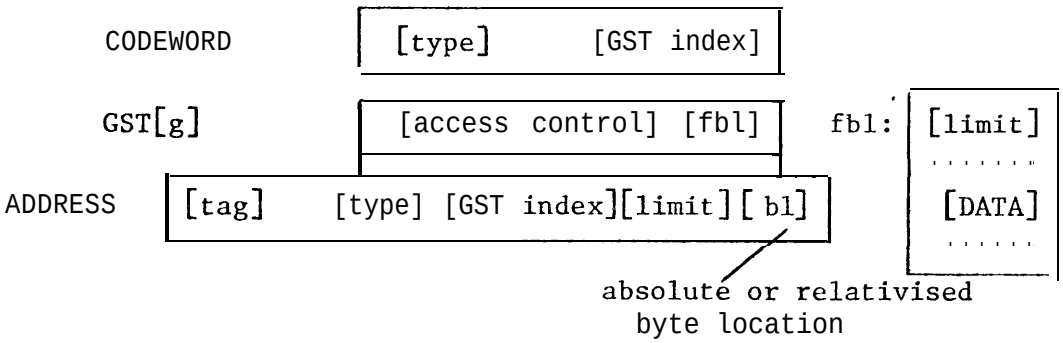



Figure 5: VCS Base Hierarchy

Resources are defined by various types of capability, found in capability segments at the branch points of the program tree. The most time-critical VCS functions are those concerned with forming addresses from segment capabilities (codewords), and with using them to access memory. For system reasons a codeword refers indirectly to store via a global segment table (GST). The corresponding address retains the GST index in order to check the accessibility and position of the segment, which happens each time an address is loaded into a register (from the PSV). The access code is used to control shared (read-only) access by several processes or unique (update) access by individuals. All such control and conversion together with the recycling of GST indices and memory is exercised by VCS microprogram, which provides a good example of the application of microcode to system problems.

The 'read', 'write' and 'modify' instructions which should strictly speaking be found on the VCS function list are too critical to handle by microsubroutine call. Users are therefore allowed to issue them directly for binary data and trusted to observe the limit and protection codes.




In the course of design numerous candidates for positions in the VCS function list have to be considered. A fundamental problem in extending the system is to achieve valuable effect without degrading overall performance. Sometimes a microcode branch is obtained 'for free', while at other times a new facility entails extra tests in a critical path. The available control store in a range of host machines has also to be considered. Options considered in that light are:

(i) selection of set elements by key rather than index value;

(ii) provision of paging facilities;

(iii) static chaining in the procedure activation list;

(iv) introduction of a third segment type consisting of a set of tagged elements;

(v) use of semaphore variables for interprocess communication.

There are many possible variations of the addressing rule such as (i) and (ii) but each entails a loss of space or time that skilled programmers will try to circumvent. The best programming environment appears to be a set of dynamically constructed, variable sized segments: they make optimal use of store and their access overheads are well understood. It is left to subsystem designers to map programs efficiently onto the tree structure, so that the store management implicit in a language such as APL is carried out in part by the language subsystem (which is aware of the details of APL usage) and in part by VCS functions which provide the containers for the APL workspaces.

VCS procedures are not intended to represent high level control structures directly, though they happen to be adequate for VCSL and simple languages such as Fortran. Recognition of static levels involves extra work in procedure management and a variety of actions dealing with special cases that could not be built into a fixed system, so it is intended that such structures be mapped by the language microcode into simulated control stacks. It seemed probable that mapping a display structure such as those found in Algol-derived languages would benefit from the ability to manipulate sets of addresses, but the practical implementations studied so far have used indirect mapping techniques, i.e. a new form of 'pointer' peculiar to the language is invented and mapped dynamically onto the VCS structures (cf the Data Qualifiers in Cobol). The advantage of such techniques is that they can take account of language parameters in the design of pointers, but we noted earlier that 20 or more microsteps may be taken to reconstruct the absolute VCS address.

Finally, various forms of semapore signalling were considered, but only a minimal 'busy' flag was implemented in the PSV. The argument against greater elaboration is that the access mechanism of the Global Segment Table already provides direct control over shared resources, associating the control variable with the resource itself, so there is little point in providing more obscure functions to the same end. The release of a segment for rescheduling at the end of a critical section is not automatic: to force it at procedure exit, for example, would again imply intolerable overheads, so an explicit VCS Release function is required.

The Variable Computer System provides support for language-oriented microprograms in easily portable form: an investment of about 8 Kbytes of microcode transfers the VCS functions, VCSL support codes, compilers, utilities, etc to a new host machine. It provides the type of support which is needed if the advantages of microcode are to be fully realised for each language, and although the function list could be improved in the light of

experience I think it is a sound method of exploiting the current generation of general purpose emulators, acknowledging that system security rests on the correct design of language interpreters.

### 4.4 Future Developments

Careful choice of words has left'the most critical question unanswered: leaving aside short-term expedients, is a general purpose host machine with two levels of writable control the best starting point for processor design? I think not, for three reasons.

Firstly, the arguments that have been used are based on measures of high level language implementation, whereas a substantial part of information processing still lies outside that well-defined area. Several systems of mediocre performance and limited applicability have resulted from the assumption that a high level language or set of languages would cover the field. On the other hand without the formality of high level constructs it is difficult to see how to make use of writable control memory.

But even accepting the limitations of high level languages it can still be argued that the interpretive approach is not optimal in many instances and that the system problems outlined earlier have still not been solved. It has to be shown that there is a better approach to language implementation with the range and flexibility of conventional systems. We begin by drawing a distinction between the inherent coding advantages of microprogrammed interpretation and the benefits which result from using fast storage or ducking behind the range architecture.

Microprogrammed interpreters have improved on fixed, complex target instruction sets to the extent that much of the redundant information in the instruction stream has been eliminated. The figures given earlier show a reduction from 500 to 200 microsteps for the average Cobol statement, or a reduction from 15:1 to 6:1 in the ratio of support steps to useful arithmetic and logic. That suggests there is still room for improvement, which might be found in a hybrid form of control in which in-line and interpretive methods can be mixed. After all, an interpreter is simply a means of calling a subroutine from the target instruction stream: its weakness is that the interpretive overhead is paid on every syllable. In other words, if we think in terms of an 8-bit function syllable, 128 codes might be assigned to hard-wired functions, the other 128 to procedure entries in a variable 'control environment'.

The starting point I suggest is that each language should be analysed from the point of view of minimising the product of microsteps and space in the representation of programs, covering both instruction and descriptor decoding. I expect, though I do not

know of a fully tested example, that the best code a compiler can produce will be a mixture of microsteps and monosyllabic procedure calls. In other words, the separation into 'interpreter' and 'target' code is no longer relevent.

The problem of presenting the control stream to the processor at high speed cannot be solved by committing the entire interpreter to control memory because it is now diffused through the program space. As it happens, it was not at all clear how to do that in a flexible manner for a general purpose multilanguage system. The conversion of 'microsteps' to 'nanoseconds' can best be treated in the broader context of speeding up memory access rates: look ahead, use cache buffers, or in the last resort pay more, but do not attempt to deal specifically with the restrictions of control memory or scratchpad. It will be noted in passing that for the multicomputer architectures envisaged the path from memory to processor is shorter than that of a centralised system with shared store highways, therefore the benefit of high speed control memory would be less marked.

Returning to system problems, we are left with (A) range cover, which it was (and still is) hoped to achieve using multiple computers, and (B) security. The dedicated-language system is not affected by the use of hybrid control: no assumptions are made about program security. The distributed-program system does depend on controlled address formation, which was achieved in the Variable Computer System by a policy of trusting the language subsystems. With hybrid control it becomes imperative to have hardware-enforced protection. It is also the case that many of the key VCS functions at present implemented by microsubroutine calls could be implemented by in-line code.

The above discussion has been based on vaguely defined 'microsteps' comparable with the vertical microinstructions of present-day machines. The reader may feel concerned at reverting to a processor style not far removed from that of twenty years ago. Is there a danger of inventing more and more complex microsteps and repeating the evolutionary cycle that led to the IBM System/360 and other 'range' architectures? The return in space that can be expected from more complex instructions depends on finding frequently repeated diagrams or n-grams that can be suitably packaged. They are more likely to occur in arithmetic, where 'hardened' floating point and decimal operation can be expected, then in control sequences. It would not be surprising to see the host arithmetic functions develop in the direction of current machine codes (with type interpretation placed on descriptor or tag gields), but the many nodes of data access appear to benefit very little from complex addressing rules.