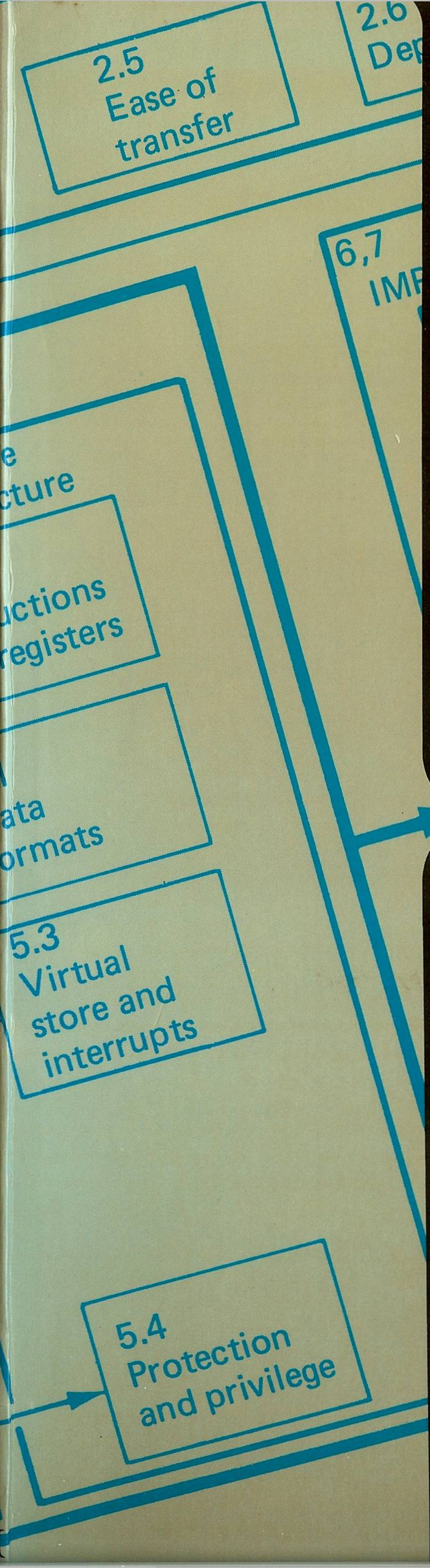


The ICL 2900 Series

J. K. Buckle

The ICL 2900 Series

[illegible]

The ICL 2900 Series

J. K. Buckle, M.A., F.B.C.S.

M

First published 1978 by
THE MACMILLAN PRESS LTD
London and Basingstoke
Associated companies in Delhi Dublin
Hong Kong Johannesburg Lagos Melbourne
New York Singapore and Tokyo

Typeset by Reproduction Drawings Ltd.,
Sutton, Surrey.

Printed in Great Britain by
UNWIN BROTHERS LIMITED
Old Woking, Surrey.

British Library Cataloguing in Publication Data

Buckle, J K
The ICL 2900 series.–(Macmillan computer science series).
1. ICL 2900 system (Computer)
I. Title
001.6′4044 QA76.8.I/

ISBN 0-333-21917-1

This book is dedicated to the other members of
the Synthetic Option Team

Contents

Preface

This book is intended as an explanatory guide for readers who are familiar with general computer concepts and wish to gain a broad understanding of ICL's 2900 Series.

The series was conceived from the outset as a range of machines and the initial design effort was concerned with those features common to all members of the range. Within this *basic range architecture* individual hardware and software systems were then developed. The present book is concerned mainly with the range architecture. Implementation matters are described when these are helpful to an understanding of general range concepts, or give the reader an insight into the way in which the architectural principles can be interpreted to meet differing criteria of cost, power and application. In addition some attention is devoted to the history of the 2900 and the influences on its design.

The book is intended to be read sequentially, but many of the individual sections are freestanding. This introduces some repetition, but does allow the book to be used for differing purposes by readers with different interests. To aid such use the logical structure of the book is shown diagramatically in figure 0.1. Each box represents an area of possible interest and the number in the top left-hand corner of a box represents the section or sections in which information can be found. Thus *design influences* are described in section 1.2. Boxes within another box indicate a logical subset of the area covered by the surrounding box. Thus section 2.1 *efficiency* is part of the general subject of *aims and objectives* and section 3.3 *program structure* is one of the *fundamental concepts*, which themselves form a part of the *2900 Architecture*. A line with a horizontal arrow-head can be interpreted as meaning 'leads to' while a line with a vertical arrow head means 'constrains' or 'controls'. Thus the *planning operations* (section 1.1) led to the *2900 Architecture* (sections 1.3, 3, 4, 5), which was constrained by *design influences* (1.2) and *aims and objectives* (2). Similarly the *fundamental concept* of a *virtual machine* (3.1) leads to an *architectural mechanism* called *virtual store* (4.3), which in turn produces a requirement for *virtual store and interrupts* (5.3) in the *primitive architecture*. The reader can use the diagram either to select areas of the book of interest to him or as a reference guide, to find, for example, all the ramifications of the *virtual machine* concept or the reasons for the *register structure* of the primitive architecture.

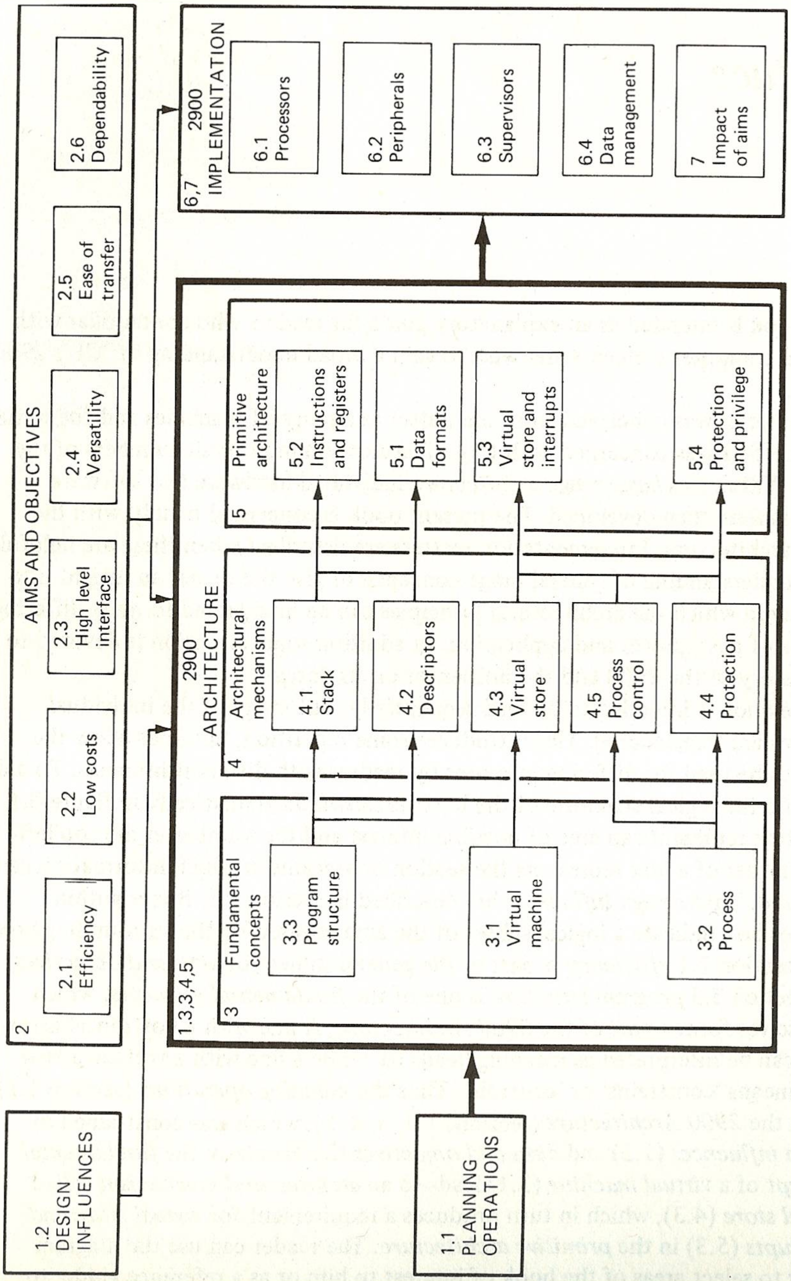


Figure 0.1 Structure of the book

Finally, whatever else the book is useful for, it is definitely not written as a sales brochure for the ICL 2900 systems. However, when one has been closely connected with something from conception, through a five-year gestation period, to birth and a year or so of infancy, it is difficult not to feel some parental affection, even after a year's separation. This can lead to a concentration on obviously attractive characteristics to the exclusion of congenital defects of a permanent or correctable nature. The author wishes to apologise for any unnecessary enthusiasm shown in the book and begs the reader's indulgence.

Crowthorne, 1977 J. K. BUCKLE

Acknowledgements

The author would like to thank the following: ICL, for the help given to him in the production of this book, and in particular E. S. Mack, Director of the Product Development Group, for permission to use the Development Group Library for research purposes; Professor Jeff Rohl, for the initial suggestion for the book; and his wife, for turning semi-literate manuscript into decent copy.

1 *The History of the ICL 2900*

The 2900 Series arose directly from the formation of International Computers Limited by the merger of ICT and English Electric Computers in 1968. This chapter is devoted to the historical background to the development. It defines the way in which the development was carried out in the new company, the influences that affected the final system design and the basic form of the final 2900 specification.

1.1 THE ORIGINS OF THE SERIES

At the formation of International Computers in 1968 it was realised that within a few years the new company would need a range of machines to replace the series currently being marketed by the individual components of the merged corporation. At that time ICT was selling the 1900 series, while English Electric as a result of a previous takeover was supplying both its own System 4 Series and the 4100 range of machines developed by Elliott Automation. The three ranges inherited by the new company were incompatible in almost every respect—word length, basic hardware structure, software construction. Although each of the ranges had their own special areas of application there was still a considerable amount of overlap. This, and the cost of maintenance of three production lines, pointed to the need for some form of rationalisation.

In addition, although there existed as-yet-unannounced developments of both the 1900s and System 4s that could prolong the life of these ranges for several years, all three types of machine dated initially from the early 1960s. Major new design enhancements would be needed to take advantage of improvements in hardware and software technology over the past decade.

As a result of considerations such as these, one of the first corporate decisions of the new company was to establish a New Range Planning Organisation. The nucleus of this organisation, known by its initials as NRPO, was formed by 'professional' planning staff drawn from the constituent companies. These provided experience both in corporate planning and market research, and in technical product planning. To this nucleus were added experts in different disciplines from all the operating divisions of ICL: development and manufacturing staff for both hardware and software, logical designers, technology experts and sales and support staff.

The resulting unit was organised into small teams, each with an individual task. The teams fell into one of three broad categories. The first category was the 'aims' teams, who were responsible for establishing the basic aims and objectives of the New Range. They carried out or commissioned market research in the United Kingdom and abroad. They determined trends in user needs and computing practice and made predictions of future development. They analysed competitive operations and produced price targets and performance criteria. They examined technological developments and evaluated their effectiveness and usefulness. And, finally, they established a corporate marketing strategy and determined the conditions that this imposed on future products.

The second category of team was the 'options' team. Each options team had the task of producing the outline design for an archetypal computer system that could form the basis of a New Range which would satisfy the basic requirements of the aims teams. The staff for each options team was drawn from as wide a variety of disciplines as possible—hardware and software designers, technology experts and marketing and planning staff. The teams were kept small—a maximum of six people—to avoid 'committee design', but could call upon the services of a wide variety of experts for advice from within NRPO, the rest of ICL and outside. Each of the teams had a specified 'option' that formed the framework for their design. For example, one team considered further, possibly radical, development of the 1900 series to meet the requirements of the 1970s and 1980s. Another team considered a similar proposition for System 4, continuing the RCA and English Electric policy of following IBM. Two of the teams were concerned with the further development of internal research projects that had been in progress for some time. One of these was a dedicated High Level Language machine, while the other considered the development of J. K. Iliffe's Basic Language Machine,[1,2,3] a prototype of which was operational in ICT's Stevenage Laboratories. Smaller teams considered the exploitation of various developments outside the company. Finally, there was what was called, rather unprepossessingly, the 'Synthetic Option'. This team had a free hand to consider any known developments or propositions inside or outside ICL and, from any elements it considered good, to synthesise a coherent architectural design. The aim was not under any circumstances to produce a compromise, and the team was at liberty, if it wished, to choose some other option or external development in its entirety. However, its chief objective was to incorporate as many compatible modern but proven computer concepts as possible in a unified design.

The final group consisted of 'assessment' teams. Their task was twofold. First they extended the requirements laid down by the aims teams, by detailed examination of particular aspects of computing with which the New Range would be expected to deal effectively. Proceeding independently of the design of the range itself, they were able to develop a set of secondary requirements that the options would have to meet. As an example, one of these orthogonal investigations was concerned with data management: the growth of data bases and the growth of techniques for data capture, data transmission and data-independent programming. Another group was concerned with 'Bridgeware', the products necessary to ease the transition of ICL's own and competitors' customers from their current systems to the New Range. The second task of the assessment teams was to establish detailed criteria by which the output of the options teams could be measured against the stated requirements, as an aid to management decision.

NRPO was not a particularly large unit and it was possible to maintain good communication between the various teams. Thus options teams were able to swap experience and ideas and to take or adapt part of each other's designs. This not only saved effort but allowed the incorporation of good techniques into an option design at an early stage, before design work had become too frozen and the Not-Invented-Here syndrome could lead to their rejection as a matter of principle. Additionally, as work proceeded on refining judgement criteria and developing the options in parallel, it was possible to compare the embryonic designs with the stated ideals. Such co-operation was encouraged and could result in option design changes to meet criteria or the highlighting of defects in the statement of a particular criterion, leading to its modification or rejection. Finally, well-established communication channels with the operating divisions of ICL, and the availability of the in-house expert consultants mentioned earlier, provided a fast turn-round for new ideas and reactions to them. As a result of this form of organisation, although the first phase of NRPO operations lasted for more than six months, during this period several of the options were eliminated for marketing or technical reasons, or were amalgamated with other similar options. At the end of the phase the remaining contenders were presented to the Company at large and were reviewed by, among others, a 'jury' drawn from senior and middle management across ICL. The outcome of these deliberations was that the Synthetic Option as defined at that time was chosen as the basis for the future ICL New Range. A number of changes and improvements were however also proposed. In particular it was suggested that the protection mechanisms should be improved to give something approaching the capability displayed by the Basic Language Machine.

Following this decision NRPO entered its second phase and grew in numbers. Refinement and further development of the Synthetic Option proceeded in parallel with an expansion of the basic architectural model to include requirements specifications for software, initial hardware models, peripheral availability and so on. More experts were drawn in from operating divisions of ICL to help with this work. By the end of this second phase of NRPO operations the requirements for the initial manifestations of the range were documented to a level at which implementation teams could be established. At this point control passed back to the main hardware and software development units of ICL and most of the technical staff returned from NRPO to work on the implementation. The reduced NRPO changed its name to New Range Organisation and changed its activities to monitoring and co-ordinating development. The development units began to design and build what were later to be called the first 2900 Series systems.

1.2 DESIGN INFLUENCES

The very definition of the task of the Synthetic Option team meant that the resulting design would be subject to a wide variety of external influences. In searching for the best modern ideas in computing science and practice, and melding them into a whole, the team were subjected to various methods and techniques and, even when these features did not appear in the final design, they had an effect on the team's thinking and its approach to solving particular design problems. As might be expected, these rather subliminal influences were more common than the complete incorporation of a design solution from another machine, since the latter almost always dragged in subsidiary requirements that were incompatible with other parts of the Synthetic Option design. Ideas were taken in to the team, thrown around among the members until the rough edges were abraded and then remoulded until they fitted into the partial design edifice that was being built up.

This process often introduced much greater generality into 2900 concepts than had been present in the original source. For example the techniques used to implement the concept of a virtual machine (described in more detail in subsequent chapters) are not new. Virtual storage was available on Altas computers in the early 1960s.[4,5] The idea of the programmer interfacing only indirectly with peripherals (perhaps via spooling) has been around even longer. Nevertheless, techniques like this have been considered more as solutions to isolated problems than as part of an over-all operational method. The virtual machine combines all such concepts into a single coherent structure which is new, but the implementation of this new structure is by proven individual techniques. Previous paging, segmenting and spooling systems are, if you will, degenerate implementations of a virtual machine system.

Many of the direct influences on the 2900 architectural design came, as one might expect, from in-house systems. For example, concepts originating in Atlas and its supervisor and developed on the large 1900s with the GEORGE operating systems[6] were well known to all the original team and provided a continuous influence on the team's thinking. Again, as already pointed out, there was considerable communication between the rival options. The Basic Language Machine in particular was very carefully studied and many of Iliffe's innovations became foundations for the 2900.

Of the competitive systems studied during the Synthetic Option design, those most relevant were probably the large Burroughs machines.[7] Although the fundamental concepts of the 2900 architecture differ considerably from those of Burroughs, similar design requirements have resulted in an over-all design 'shape' that is quite similar. The Burroughs approach of a single high level language machine was rejected for 2900 but the latter's requirements for good all-round high level language performance led to the selection of some similar implementation details. In the operating system area several MULTICS[8] concepts had a considerable influence on 2900, particularly in the sphere of protection where, even seven years on, it still represents the state of the art.

However, the single most important external influence on the 2900 architecture was almost certainly the Manchester University MU 5 system.[9,10] This is hardly surprising. The association between Ferranti Computers, a constituent of ICL, and Manchester University went back to Mark 1 and had progressed through Mercury and Atlas. At the time of the establishment of New Range Planning Organisation a team at Manchester University had been working on the design of their fifth machine for some time. Indeed, one of the options briefly considered for the new ICL range was to adopt the MU 5 architecture intact. This approach was ruled out since the objectives of the two developments were not identical, but sufficient similarities were present to make the Manchester work of considerable interest to the Synthetic Option team. Two aspects were particularly important. First both teams recognised the need to generalise the supervisory software system while improving its efficiency, and the decisions taken by the University in this area were well received by the ICL team. Second both teams were aware of the ever-increasing trend towards use of high level languages rather than assemblers, and the research and design done at Manchester were of great use in the formulation of the 2900 architecture in this area.

Although there are fundamental differences which mean that MU 5 falls outside the 2900 Range Definition, the early co-operation between the two design teams ensured that there would be a family resemblance between the two systems. Many of the fundamental approaches to storage management and process structure on 2900 are directly derived from the MU 5.

1.3 BASIC 2900 ARCHITECTURE

As already stated, the 2900 Series was conceived from the start as a range of machines that between them would cover a considerable spectrum of power and facilities. This made it necessary to define the new series in terms that were of general application to all the members of the range and not merely to design the first few models that were to be developed. In addition we were concerned not just with the hardware machines that would form the range but with a number of hardware-plus-software *systems*. The range definition therefore had to be system-oriented and to cover items that might be implemented in different ways—purely by hardware, or purely by software, or by some mixture of the two—on different models of the range.

The description of the range in this fashion forms the basic architecture of the 2900 Series; it includes all the most important and interesting features and relationships of the components of a 2900 system. In particular it specifies the criteria by which a particular hardware–software system can be judged to be a member of the range. Because of the range and system concepts underlying the 2900 these criteria are not always what one might expect from older range definitions. For example, although all the early members of the 2900 range share the same order code, the order code definition is not a part of the basic architecture. Because the hardware–software system is considered as an entity, changes in the instruction set between models may be hidden by the intimate software and so be invisible to all normal users. On the other hand considerations of data storage and interchange, as well as international standards, mean that the data formats do form part of the basic architecture: a 2900 is an 8-bit-byte, 32-bit-word machine, for instance. These considerations in turn place restrictions on the features necessary in the instruction set, without actually defining its contents.

The basic architectural description of a 2900 forms a complex hierarchy. At the top is the *architectural model*, which is a general description of a 2900 system. In producing this model, the feasibility of actual implementations was of course considered but the model is kept as simple as possible and any specific 2900 system design has to be a practical realisation of this. The architectural model is, as one might expect, most concerned with aspects of the central processors, storage mechanisms and lowest-level supervisory software, together with the interfaces with the other hardware and software components that go to make up a total system. These components are in turn described in the next level of the hierarchy: how peripherals fit into the architectural model, supervisor architecture, communication systems, multiple-processor operations and so on.

Many aspects at this second level, particularly where they interface with the architectural model, form *range standards.* Other aspects, however, are not so fundamental and can best be described as elements of architecture for a sub-range within the over-all 2900 Series. The lowest-level documents within the hierarchy describe in detail the implementation of specific pieces of hardware and software that conform to the specifications of the higher-level documentation. Implementations of these items can be combined, again according to rules specified at the higher levels, to form actual 2900 systems.

The actual design of the 2900 did not, of course, proceed in the strict top-down sequence outlined above. Even before the Synthetic Option had been chosen as the basis for the new range of machines, detailed investigations of much lower-level hardware and software technology matters were proceeding. The results of these had an obvious effect on design decisions at a higher level. Again, at a very early stage the Synthetic Option team themselves were forced to sketch out designs for specific systems to convince themselves that such implementations within the architecture were in fact feasible. The over-all 2900 design thus proceeded in an iterative fashion through the levels, but with the centre of the iteration descending to lower levels as time went on.

While this hierarchical description is the best form of reference documentation for the 2900, it does not form a very convenient method for an expository document like the present book. Some of the higher levels of abstraction are difficult to grasp without specific examples, while some of the intermediate-level range standards, such as the status of peripherals in the architectural model, are too detailed to cover in a work of this size. For this reason the remainder of this book is structured rather differently. After chapter 2 outlines the basic aims and objectives determined by ICL for the New Range, the fundamental concepts of the basic architecture are first described in chapter 3. Chapter 4 then describes some basic architectural entities which are a mixture of basic architecture and range standards. Chapter 5 defines some important second-level or primitive architectural features that are again a mixture of range and subrange standards. Finally, to place the preceding descriptions in perspective, chapter 6 gives some details of the first implementations of the 2900 Series that have been produced.

2 *Aims and Objectives*

The last chapter described how the aims and assessment teams of NRPO specified the guidelines for the design and development of the 2900 Series. These took the form of a catalogue of requirements, which were of varying importance and seemed at times to the Synthetic Option team to be too numerous and in some respects self-contradictory. In retrospect, however, it can be seen that the requirements do indeed form a coherent set. Not all affect the architectural model, some being more concerned with implementation details.

The requirements can best be considered as a hierarchy that has at its apex the needs and wishes of the user of the system. This hierarchy is set out diagrammatically in figure 2.1. ICL was out to make a system that would be financially

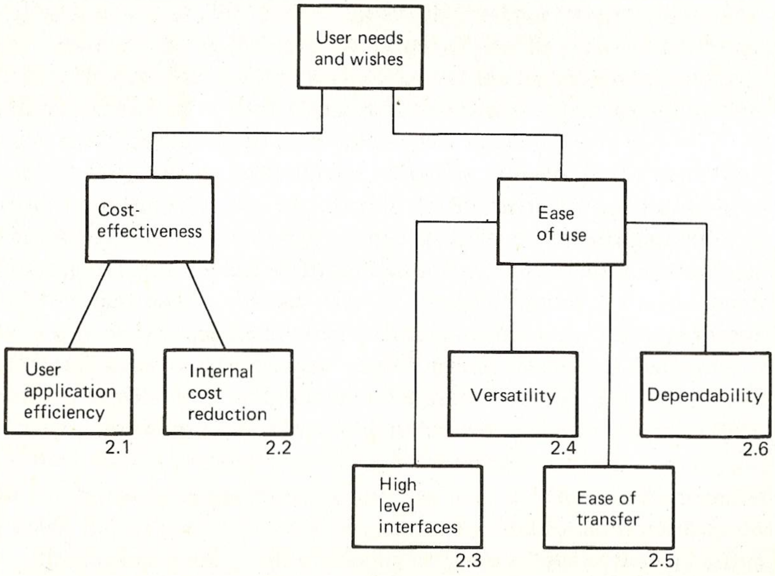


Figure 2.1 Design constraint hierarchy; numbers under boxes denote relevant sections in text

successful and it was realised that modern customers were not prepared to take an off-the-shelf system, however good, and adapt it to their needs (or, worse, to adapt their needs to it). The basic requirements for the New Range were therefore specified by the way in which its future customers wished to use their computers. The projected customer base was not homogeneous, and even within one customer installation the needs of the management, technical staff and end users were liable to be different. However, strong trends could be seen. At the grossest level the users' needs split down into *cost-effectiveness* and *ease of use.*

Cost-effectiveness, the ability to get as much as possible for your money, is a well-known business desire. There are two possible approaches to providing a cost-effective system, both of which were taken into account in the 2900 design. The first is to reduce the cost to the customer by reducing ICL internal costs. The second is to make the end systems as efficient as possible in areas of importance to the user.

Ease of use is much more complex, but again broad areas of agreement over a wide user spectrum could be discerned. First, users wanted their dealings with the computer to be at as high a level as efficient use permitted. People are a valuable (and expensive) commodity and cannot be allowed to waste time adapting their statements of problems to forms that the computer finds acceptable. There was thus a need to define computer systems from the user levels inwards and not from the hardware outwards. A closely connected requirement was that of versatility, both in the variety of applications and in the way in which they were being operated within the system. Thirdly, it was recognised that the investment in existing software and hardware is considerable in most installations and it was necessary to provide protection for this investment. Cheap and efficient methods of using existing hardware, software and data were required. Lastly, users required a dependable system. Since much of the day-to-day operation of a large business is tied up with the computer, users simply could not afford to be let down.

The remainder of this chapter considers these block requirements—efficiency, low internal costs, high level interfaces, versatility, ease of transfer, and dependability—in more detail, and highlights some of the effects that each had on the design of the 2900 Series. It can be seen that there is considerable overlap in many of the requirements—for example, hardware assistance for high level language compilers in turn helps provide more effective TP systems, and ease of use must always interact with effectiveness.

It can also be seen that considerably more space is devoted to high level interfaces than to any other section. This is not because it is considered of greater importance but because, while other user requirements apply almost directly to the system design, the requirement for high level interfaces needs one further level of indirection. It is necessary, in order to understand the design constraints, to establish what the mechanisms that provide the high level interfaces themselves require of a computer architecture. Section 2.3 attempts to do this.

Each of the following sections is a statement of particular objectives for ICL's New Range designers, against which their success was to be measured. Orthogonal to all these was the requirement—less explicitly stated but no less important—that the proposed solution be a first-class one from a technical point of view. It was of paramount importance in meeting these requirements that a coherent architecture should emerge. Individual solutions to most of these problems exist in various systems already commercially available. The requirement for 2900 was to combine these solutions in such a way as to produce a race horse rather than a camel.

The way in which these objectives eventually gave rise to the 2900 series is far too complex a story to be recorded here. As might be expected it involved much iteration and several false starts. Also the mapping of requirements on to design concepts is not easy, since any concept may help meet several requirements and, conversely, each requirement may affect widely different areas of design. The following chapters therefore omit the intermediate proposals and the long and painful task of coming up with a complete system architecture. Instead they describe the design of the 2900 that finally emerged and was implemented in the 2960, 2970 and 2980. In describing this architecture, mention is made in passing of some of the requirements of this chapter and how they are satisfied, but in general the architectural description is based on more fundamental concepts of processes and virtuality. However, in order to tie up loose ends a section of chapter 7 is devoted to explaining at least some of the ways in which the eventual design meets the more important objectives set for it.

2.1 EFFICIENCY

The efficiency of computer systems is a much debated subject but it is still an elusive entity and difficult to measure. Early measurements of performance, such as addition speeds, have long become meaningless and even weighted instruction times or measurement units such as the Post Office Work Unit (POWU) are largely irrelevant in the face of very complex user applications. In many business applications the speed of the central processor may be entirely irrelevant, or only relevant in terms of its effectiveness at running the operating system. Efficiency to the user is normally concerned with the speed and ease with which his own particular problems can be solved and this is often affected more by data speeds and high level software effectiveness. These matters are dealt with later under high level interfaces.

Nevertheless, the speed of basic hardware is by no means irrelevant. For example, there are still applications, mainly of a scientific or near-scientific nature, to which pure number crunching is important. What is of more general importance is that the speed of basic manufacturer-supplied software, which greatly affects over-all work throughput, may well be dependent on the speed of simple data-manipulation instructions. The basic design of the 2900 Series assumed that the early models would use current tried technology but the general design principles must permit the exploitation of possible future developments. In this context it is perhaps interesting to note that after the 2970 had been designed, the hardware-development organisation of ICL was able to build, quite quickly, a number of machines that were logically equivalent to a 2970 but were constructed entirely using old-fashioned (and cheap) 1900 technology and peripheral equipment. These machines, known variously as 'hardware simulators' or 'architectural prototypes', were used to develop engineering and user software before the availability of production machines.

Again, although logic and storage costs were both falling, and had been for a decade or more, ratios of cost between mass storage, main storage and 'registers' were not closing as fast as some forecasters had predicted. The architecture thus still had to allow for different mixes of such components in various range members. In a similar way, the use of pipelining and instruction-overlap techniques had to be permitted at the top of the range without prejudicing the performance of smaller 2900 models that could not afford such luxuries.

While most of these requirements affected design levels lower than that described by the architectural model, the Synthetic Option team and its successors had constantly to check that no decisions they took prejudiced the realisation of these objectives.

2.2 LOW INTERNAL COSTS

In order to keep the cost of 2900s to the customer at the lowest possible levels, ICL was keen to minimise its own costs. This applied both to the investment to be made in hardware and software development and in the manufacturing costs of individual systems. The means adopted to achieve this end were basically similar in both hardware and software areas and can be summed up in the two words *modularity* and *standards.*

In software it was essential to allow for the greatest possible modularity using standard software production techniques. As well as the well-known and unanswerable technical arguments in favour of such an approach to software implementation, ICL had an enormous task to produce a software system that would be competitive. It had to produce, from the outset, a set of software that would be at a similar level, at least in terms of facilities, to the software catalogues on existing machines, which had been built up over several years. It was thus necessary to carry out a complete outline design of the projected software and then implement selectively so that individual components could be improved, replaced or added at a later date without collapsing the whole structure.

In hardware, modularity was again a well-known cost-cutting technique, and was adopted from the outset. However it was necessary to maintain flexible interfaces in the design so that particular 2900 models could exploit new technology as it became available. As a particular example of this the interface between hardware and software itself is deliberately kept flexible. It has already been stated that there is no range-defined order code but, even where two models of the 2900 Series share the same order code, individual instructions may be implemented in pure hardware, microcode, or by software extracode.

2.3 HIGH LEVEL INTERFACES

By the end of the 1960s the computer had ceased to be a technological wonder and had become just another tool, although an important one, in the everyday operations of businesses and scientific and research establishments. With this process it slowly came to be realised that the man–machine interface was far too close to the machine. To perform even relatively simple jobs required programmers and operators with a great deal of knowledge and technical skill in the particular hardware and software of the customer's installation. As computer usage increased such people became scarce and expensive. Accordingly, throughout the decade, there was a tendency to raise the level of the man–machine interface and to make system input as close as possible to the level of the natural statement of the problem or the natural form of the data. Two distinct aspects of this can be seen. The first is the almost universal adoption of high level programming languages such as COBOL and FORTRAN in preference to assembler. The second is the high-level treatment of data, both in the movement towards integrated data management systems and data bases, and in the idea of capturing data where it occurs and delivering results directly to where they are needed.

Such developments eased the problems of scarce staff and long development times but introduced new user worries in terms of efficiency. A machine that has been designed by assembler users for assembler users is unlikely to be well suited to running structured high level languages. Beautiful bit-twiddling instructions cannot be exploited by high level languages without large and complex (and hence slow, bug-ridden and expensive) optimising compilers. Again, while logical designers think in terms of words and bytes as data items, the handling of arrays, structures and records is unlikely to be as good. In the data handling area the problems were similar. Data management and communication subsystems tended to be grafted on to existing hardware–software systems. The resulting 'layers', while providing some sort of structured modularity, lead to unnecessary communication problems between the subsystems and to duplication of code and effort in each. This in turn has an adverse affect on the system performance.

The task set for the 2900 designers was to overcome these difficulties by starting from the outside with the user-interface requirements, and working inwards towards a sympathetic hardware and basic software design. The data-related aspects led to specific requirements on the hardware and supervisory system—the need to provide at the basic architectural level

- an efficient but totally protected communications environment
- inherent ability to handle a wide range of communications equipment
- fast interrupt handling
- efficient handling of basic data-management functions
- the ability to handle data and code in as independent a fashion as possible
- an effective matching between the design of the hardware and the basic supervisory system.

These requirements had a fundamental effect on the 2900 design as can be seen in later chapters, contributing to the adoption of virtual machine processing, a hardware stack and descriptor addressing. Some are dealt with in more detail in section 2.4 below.

Since the 2900 is sometimes said to be a high level language machine it is perhaps worth investigating in more detail the language-imposed constraints on the basic architecture. The idea of building a machine that was designed specifically for one high level language was considered in NRPO; indeed one of the original 'options' was such a high level language machine that would take the same approach as Burroughs[7] but probably with a language other than Algol 60. This was rejected at an early stage because of the large investment in programs written in a wide variety of existing languages throughout the computer community. While COBOL is undoubtedly the most widely used language, with FORTRAN close behind, both Algol 60 and PL/I have their adherents and even more minority languages are well entrenched in certain areas. Investment in programs in all these languages is now so great that users are often close to the state of the telephone companies: although new technology that could increase efficiency by an order of magnitude is available, the cost of replacing existing stock and retraining staff is too large to allow such a revolution to take place. Likewise, a large percentage of the computer market is financially tied to an existing programming language. Selection of the language on which the new machine would be based would therefore immediately rule out large groups of potential customers. Instead, the designers were faced with the more difficult job of producing a computer that would be better than a conventional machine over a wide range of languages.

Analysis of the object code and compilation techniques of a wide range of compilers and languages produced a number of desirable attributes for a system architecture that were not language-dependent. By implementing these features it would be possible to produce a compiler that could generate code of greater efficiency at the same cost as on a conventional machine (or code of the same efficiency at less cost). This is shown diagrammatically in figure 2.2. The vertical axis represents the production effort needed to obtain a compiler of a particular efficiency. This is normally directly proportional to the compiler size and complexity and, of more interest to the user, inversely proportional to the compilation speed and the compiler reliability. The horizontal scale represents percentage efficiency of the object code compared to optimum hand-coding over small code segments. (Note in passing that the object code comparison over large segments will be better, even on a conventional machine, since by using a high level language the programmer can concentrate on the over-all effectiveness of his algorithm much more than a hand-coder can.) As one approaches zero efficiency or 100 per cent efficiency, the type of architecture is immaterial and the compiler production effort tends to a constant value, or an infinite value, respectively. However, in the middle ranges adoption of the attributes defined below can provide either savings in effort or increases in efficiency. The attributes are, not necessarily in order of priority

- a 'clean' order code
- good procedure handling
- good structure handling
- ease of expression evaluation
- an efficient object-code support environment.

These are dealt with in turn below.

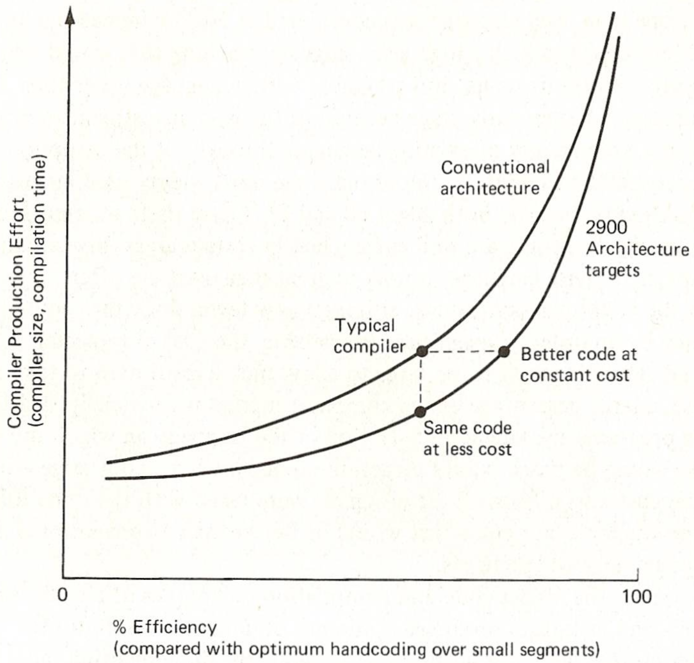


Figure 2.2 High level language aims

Clean Order Code

Wirth[11] has pointed out how the lack of regularity in an order code becomes apparent when one tries to design a language around it. The same problem arises when trying to map an existing, regular language on to an order code. 'Holes' in an order code, such as the ability to reverse divide in floating point but not fixed point, or to have only fixed-point immediate operands, either complicate the code-generation algorithms out of all proportion to the hardware savings or mean that some useful features of the hardware cannot be exploited.

To be useful for a compiler writer, an order code must be regular in two respects. First it must be regular with respect to data type. High level languages normally allow identical operations on all permissible arithmetic data types: integers, floating point, decimal, single or double length. The hardware should similarly provide the same repertoire for all data types wherever this makes sense. Secondly, order codes normally provide several operand types: operands in store or in registers, operands accessed indirectly via a register or store location, immediate operands contained within the instruction itself. If these are to be effectively exploited by a compiler they should be applicable to all instructions, not just an arbitrary subset, and to all data types. An order code with these two types of regularity, sometimes called an orthogonal instruction set, is a prime requirement for lowering the compiler-effort–efficiency curve.

A natural spin-off from an orthogonal instruction set is a reduction in arbitrary hardware limits. It has been said that in computing there are only three good numbers—zero, one and infinity. Hardware built for assembly coders on the other hand seems to prefer numbers like 8, 16 and 128. Careful attention to compiler needs at an early stage means that, where limits need to exist, they do not make life unbearable for the compiler designer. Finally, hardware designed by engineers for engineers often contains what one might call *bit-twiddling instructions*: orders that exploit peculiarities of the hardware but are of only very limited use. Providing this use is of importance to the over-all operation of the system, this is no problem but one must realise that such instructions cannot normally be used by a high level language compiler. Considerations of what compilers can use will often lead to a less efficient but more general form of instruction that is of much greater value to the over-all system efficiency.

Procedure Handling

One tends to associate procedure handling with Algol and its derivatives but the basic subroutine concept pervades all programming languages. Even a non-procedural language such as COBOL contains primitive, explicit subroutine mechanisms and makes even more use of procedures implicitly by library calls. The requirements for efficient handling of such relatively simple subroutines are fundamentally the same as for more advanced languages. Admittedly FORTRAN or COBOL may not require recursion, but the same is true of 98 per cent of all PL/I and Algol procedures. However, where recursion provides a natural solution to a problem, any other solution tends to be either grossly inefficient or totally opaque.

With the growth of complex operating systems and transaction processing, the interfaces between the controlling system and the high level language application program itself are of equal importance and these often show the same characteristics as a procedure call in one of the more modern high level languages.

What then are these characteristics? Most important is that the invoked procedure (or program) requires work space of its own. This is of three separate types. First, there are the parameters that are passed to the procedure for use in this particular invocation. Second, there are the named data items ('names' for short) that the procedure declares and uses. Third are implicit working variables, which occur as partial results during the evaluation of the steps of the algorithm that the procedure represents. While it is possible in many cases to assign this work space long before the procedure is invoked, this is not normally an efficient use of resources, particularly in transaction processing or real-time applications.

A requirement of good procedure handling is thus that the basic architecture should handle assignment and reassignment of storage space for names, parameters and work space in a simple and efficient manner. This mechanism should be capable of dealing with the most complex types of procedure and very efficiently with the simpler, more common types. Recursion is necessary in some circumstances and it should also be possible to support code that is reentrant or shared between different applications simultaneously. Such support must not however be allowed to degrade the efficiency of simpler procedure types.

The possible complexity of procedure definition can with conventional computer design lead to high overheads on the actual call of the procedure and exit back to the caller. With modern programming techniques it is becoming vital to reduce this overhead, since programs are tending to include more procedure calls than before. The splitting of application programs into many procedures is now seen to be preferable to the construction of monolithic programs, since this produces better-defined, better-structured programs that are easier to debug and maintain. For these benefits supporters of structured programs are prepared to pay some penalty in efficiency, but it is obviously important to reduce procedure call and exit overheads as much as possible in the basic system design.

Structure Handling

Considered from the point of view of access, program data is essentially of two, and only two, types.

Scalar data items are those that correspond directly to individual units of storage (or small collections of such units). The variety of subtypes of scalar data depends on the particular language—real, integer, character, logical, bit, Boolean, fixed, complex, double-precision. The uses of the subtypes and the arithmetic or logical machinery for manipulating them may vary widely; however, as far as storage is concerned, they should be accessed and stored in the same fashion. It is with such scalar items that the clean order code and addressing functions are essentially concerned. In programming terms the scalar items are referred to individually by name.

Structured data items, on the other hand, are collections of scalar items that are treated in some sense as a named aggregate. While a few languages provide facilities for dealing with these aggregates as entities (APL, for example), in general the individual elements are accessed and manipulated by some form of indexing, slicing or mapping carried out on the named aggregate. The complexity of the aggregate depends on the language used and the form of application. With the older scientific languages one tends to think mainly in terms of arrays, and with commercial languages in terms of records and character strings. More modern languages have generalised these concepts to provide hierarchical structures that can themselves contain arrays, strings or other structures as substructures.

Some special languages contain even more esoteric structures—lists, trees, etc. These, and the need for efficient and convenient access to hierarchic structures, give rise to a new scalar type—reference or pointer variables that can be used to access other variables indirectly. Such scalars can of course themselves appear in structures, giving the possibility for extremely complex addressing patterns.

In a 'high level language machine' there is thus a need to deal not only with the common basic scalar types but also to assist as much as possible in the access of structured elements. There is also another requirement—to assist in detecting access errors. With complex structures there are many possibilities of error in calculating addresses. For example, array subscripts may be outside the array bounds or one may treat a character string as an integer by misreference. Not all of these errors can be trapped at compilation time, and run-time checking code normally brings heavy overheads in time and space. As a consequence such checking code is normally used only during debugging and removed for 'production' runs. However, as has been pointed out many times, no program is ever completely debugged and, particularly in high reliability systems, such checking is really needed at all times. The only solution is to provide some form of hardware checking that can be done efficiently by overlapping it with other hardware operations. Such considerations become even more important when the structured data is in some sense self-defining, for example, in data base manipulation languages.

Expression Evaluation

In the past considerable attention has been paid to efficient ways of translating complex arithmetic expressions into machine code. Less attention has been given to the design of the machine code to make this process easy for the compiler. More recent investigation of real programs however (for example, by Donald Knuth[12]) has shown that both studies tend to be mainly of academic interest only, since the majority of expressions used in practice consist only of one or two elements. It would nevertheless be nice if the 'high level language machine' did assist the compiler in this area.

Of much more importance is the need to optimise usage of hardware features in object-program code. Take the example of optimisation of register usage. Compiler algorithms for such optimisation, although well researched, are still not simple and in any case some general-purpose registers need to be permanently or semi-permanently dedicated to specific purposes by the compiler. This wastes the generality of the hardware, but the interface between the hardware designer and compiler writer in a conventional machine does not allow the designer actually to take advantage of such specialisation.

An associated problem is concerned with the access of data elements. While the hardware designer provides a completely general store, which the compiler then structures into instruction store, temporary data, scalars and structures, neither hardware designer nor compiler writer can take advantage of the specialisation of data areas. The compiler writer cannot have hardware checks that data or instructions are not being misused and the hardware designer cannot optimise his data access techniques. A cache store, for example, which tries to reduce store accesses by saving the last *n* accesses to main store for possible re-use, must at best be a sledge-hammer-for-a-nut mechanism. Any high level language programmer knows that, if he uses the scalar *a*, there is a high probability he will use it again soon, whereas if he uses the array element *b* [*i*] there is a much lower probability of re-use, but a high probability of an access soon to a different element of *b*. The cache-store designer has no real way if distinguishing between these uses and the hit rate for items held in the cache store cannot be expected to be very high.

The hardware designer must therefore have information about the way in which object programs will use his hardware to allow him both to assist such usage, and to take advantage of it in his design.

Object-code Support Environment

In thinking of a high level language machine one is apt to consider only the efficiency or ease of production of the code that the compiler actually generates. This is only part of the story. To begin with, the execution of the high level program will in addition involve code not generated by the compiler directly—library routines and operating system code. To the compiler these items are logically equivalent: they are a method of obtaining a service and can be considered as identical to the call of a high level language procedure. In practice, however, the mechanisms are usually different. Not only may the linkage to a library routine be different to a compiler-generated procedure, but operating system services are obtained by extracode or macro-instruction. Such unwarranted variation leads to unnecessary object-code overheads and/or undue compiler complexity.

A particular example of such complexity occurs with input/output and file handling. Operating systems tend to provide either extremely primitive access mechanisms, which each high level language must use to build up its own I/O procedures, or high level access procedures, which may mask complex data management routines but do not meet the requirements of any particular programming language. What is needed is a range of access mechanisms from direct data transfer to sophisticated data base handling which can be called in a similar way by the object program, with mapping information possibly being provided separately in the job description for a particular run. In other words, except for high level languages that specifically deal with data-access mechanisms, the specifications of the mechanisms should if possible be orthogonal to the production of the object program by the compiler.

Another aspect that is lost if we consider only the object code of a high level language program is the way in which the total program—code, data, work space—is mapped on to real storage. If each compiler is forced to take its own decisions on such mapping, particularly with regard to overlaying code or data or both if the total size exceeds the available mainstore, not only will there be unnecessary duplication but we will again forfeit the possibility of direct hardware assistance, since each language will inevitably implement differently. Such services need to be provided in a standard and straightforward fashion for all languages by the operating system.

What is of even more importance is that handling overlaying in the compiler or even the loader means that the overlay structure is fixed before execution. While this may be optimised for the program, the technique cannot take into account over-all system resource optimisation with a dynamically varying work-load. Such a partition technique is inevitably wasteful of resources when considered from a system viewpoint–local optimisation produces, if not global system chaos, at least global system inefficiency.

If we extend our view further from individual processes to total system operation a third need arises, that of protection. We have already seen the need for hardware help to prevent corruption of internal process data. It is also vital that separate programs should not be able to interfere with each other. In applications like transaction-processing systems where separately compiled programs are accessing common data, possibly simultaneously, such protection is vital and cannot be provided by individual compilers. It must therefore be a service for all programs that is incorporated into some combination of the hardware and basic system software.

These then are the basic requirements for a good high level language engine. Note that not all are direct hardware design requirements. Some are requirements on basic operating system design, which, in turn, may themselves give rise to hardware requirements. It is worth noting in passing that many of the requirements of high level language programs are very important to the production of operating system software itself. It is the lack of good support for high level language features in existing machines (Burroughs being a notable exception) that has been responsible for the regrettably low usage of high level languages for the production of system software. An operating system is, in effect, a large collection of independent and asynchronous processes and has sections that may be obeyed many times a second; in these circumstances, such things as error protection and procedure-entry efficiency become vital concerns. Unless the architectural design helps in these areas the only choice is between relatively efficient but unstructured and bug-ridden systems written in assembly code, or maintainable but inefficient systems written in high level languages.

2.4 VERSATILITY

It was obvious that with a large and varied customer base and potential market, any new ICL development would need to be versatile. There were two important aspects of this, which were largely orthogonal–versatility of applications and of modes of use.

As far as applications were concerned the new series had to be capable of dealing with both scientific and commercial requirements. To begin with, this means support of both types of high level languages–a matter already dealt with in section 2.3. It also meant the ability to provide on individual 2900 installations the basic configuration requirements that each type of application considered important—for example, to allow an individual customer to spend his money either on processing speed or peripheral connectivity. In practice, though, the boundary between scientific and commercial applications had become considerably blurred over the past few years. For example, scientific installations were beginning to make heavy use of data bases—a facility previously thought of as a commercial concern. Conversely, commercial computer owners were using their machines for forecasting and analysis purposes and therefore using programs with a high scientific content. It was thus important not merely to provide a straight choice between commercial and scientific configurations but to allow the customer to buy a balanced set of facilities that provided the best match for his work profile.

Whatever applications the customer wanted it was also important to allow him to operate the computer in the most convenient way—versatility of modes of use. Batch processing, the original way of initiating work, was still much in use and unlikely ever to die out, but was rapidly being supplemented by remote job entry, multi-access computing and—perhaps the most important commercially—transaction processing. These modes of use placed different requirements on the system.

For good batch processing the essential requirements are high over-all throughput, good control of jobs and optimum use of computer resources. In meeting these, the processing times for individual jobs, while important, were secondary considerations. The fundamental process was the efficient scheduling of work and resource employment. Remote job entry (RJE), the ability to insert jobs into the batch stream at several, possibly remote, points, meant geographical convenience for the user. For the system, it imposed rather more complex scheduling requirements (for example, output normally had to be returned to the original source of the job) and the need to handle effectively communication channels. With transaction programming (TP), a rapidly growing operational method, the emphasis is quite different. Here the over-all operational efficiency is subordinated to user convenience. Looked at another way, the scheduling has to consider the user's time as the most important resource. A user transaction, which normally requires execution of a fairly simple application program, demands a response in 'real time'. There are thus a large number of small jobs to schedule and, since they are to be run on demand, the order or combination in which they run is indeterminate. To handle such operations the hardware and basic software system requires a number of characteristics that are not necessary for batch or RJE—the ability to handle a large number of small jobs; the ability to use the same simple job in a multiplicity of contexts; and fast job switching, which in turn requires fast interrupt handling. In addition the communications-handling requirements of RJE assume even more importance.

Between batch and transaction processing but slightly at a tangent to them comes multi-access computing (MAC). Here we are concerned with many users sitting at teletypes or videos and interacting with the computer to edit files, develop programs or operate complex systems, for example, computer-aided design systems. The fundamental difference between TP and MAC is possibly that, while the former requires for each transaction the complete execution of a simple program, MAC users are normally communicating with large complex programs, like compilers, several users may be using the same program at the same time and a user–machine interaction is only a step on the over-all operation of the program and not a complete execution. It can be seen that while interrupt-handling and communication requirements between MAC and TP may be similar, optimum scheduling and work-organisation techniques are normally quite different.

Consideration of user needs in these areas does not, however, allow us to choose completely between the various modes of use. Very few modern organisations are operated on a pure batch basis. Nearly all will have some part of their workload in the form of RJE, TP or MAC. Similarly even what are thought of as pure TP organisations will normally have batch runs for bulk input, report production or housekeeping operations either outside the normal transaction hours or as low-priority jobs during transaction processing. It is thus necessary to allow the customer to mix his own cocktail of the various operational modes to suit his particular needs.

There is a third aspect of versatility, which applies not to any particular 2900 implementation but to the 2900 definition itself. Since the 2900 was envisaged as a range of machines covering a wide spectrum of power and performance, the chosen design had to be capable of implementation at both ends of the range. Nevertheless a choice of a definite target area had to be made as a guide to optimisation decisions. Difficulties with the 1900 Series had been encountered because the first manifestations had been at the low end of the range and ICL experience had conclusively proved that, within limits, it was easier to develop a less powerful system from a more powerful one than vice versa. Additionally, it was realised that improvements in technology and reductions in costs meant that the mean performance level of computer installations was constantly rising. The chosen target point was therefore towards the upper end of the range, somewhere between the eventual 2970 and 2980 implementations. Nevertheless every design decision taken was vetted to ensure that it could be effectively implemented in both less powerful and more powerful systems.

2.5 EASE OF TRANSFER

It has already been pointed out that potential customers for the 2900, whether ICL or competitors' users, could be expected to have considerable investment in their existing installations. This investment would take the form of existing hardware, staff training and programs. The first two of these can be taken care of in a straightforward (if not simple) manner by the adoption of standard hardware-interface techniques and international standards wherever these were possible. This would allow maximum use of existing peripheral equipment with the 2900 and minimum staff retraining.

Protecting investment in programs is rather more complex. The ways in which programs can be transferred to new systems form a spectrum. At one end the original source code can be amended where necessary and recompiled on the new machine; at the other it may be necessary actually to execute the alien machine code on the new machine, a process generally known as emulation, following IBM's original provision of this facility of 1400s on the 360 series. The point in the spectrum that is chosen by a user for a particular program depends on a number of factors: whether the source language is supported on both machines, if the program has a limited or indefinite predicted life, if it is subject to constant amendment or frozen, etc. A user may use more than one transfer method over a period of time; for example, he may use emulation to allow rapid transfer to the new system to begin with and then transfer and possibly amend the source code to take advantage of new facilities at a later stage.

In general all program conversion can be done by architecturally independent software tools so that only the requirement for emulation has an effect on the basic architecture. Two basic methods of implementation of emulation had to be taken into account. If a particular 2900 processor was to be microcoded, it would be possible to use alternative microcode sequences to provide the order code of the emulated machine. The ability to do this would obviously impose constraints not only on the microcode itself but also on the basic system software which must be capable of dealing with more than one order code. Alternatively, particularly on non-microcoded machines, it would be necessary to attach an alien-order-code processor to provide emulation. This is effectively a requirement for the 2900 system to be a multiprocessor system. As we shall see this requirement arose from other considerations but here we have an added requirement that the configuration need not be symmetric with each of the processors identical, and in fact it may be necessary to have several processors with different order codes.

2.6 DEPENDABILITY

With the computer system becoming an essential factor in day-to-day operations of a customer organisation it is essential that he can depend on it. There are a number of facets to this. First he would like it to be *reliable,* that is, not to break down for either hardware or software reasons. This must be a major aim but, since nothing in this world is perfect, an ultimately unrealisable one. Since an average user would not consider it cost-effective to pay for the reliability required by, say, a space mission, he must expect some failures. In such circumstances he would like the system to be *resilient*; capable of easy recovery, preferably automatic, or of 'fail-soft' operations if not 'fail-safe'. Finally, he would like the system to be *secure*; that is, various hardware and software elements of the system should be protected from each other, and error propagation throughout the system should be prevented. System components should be designed to be mutually suspicious. Such security is particularly essential in transaction-processing situations, where individual programs or hardware components must be prevented from corrupting the system.

Methods for providing reliability, resilience and security are diverse but overlapping. It must be possible to meet the other requirements of the system within the limits of known technologies. Reliability requires at least the possibility of redundancy in areas where an individual user feels this to be important. Resilience requires the possibility of reconfiguration and, in particular, of multiple-processor configurations to allow graceful degradation in situations where this is important. In order both to recover easily and to prevent error propagation it is necessary to trap errors as soon as possible; thus hardware-error checking and diagnosis becomes a requirement. Error propagation can only be prevented if there exist well-defined boundaries at which the error can be caught; in other words modularity of design of hardware and software is essential. Finally, a hardware-assisted protection system is needed for software to provide the same level of security as error checking provides for hardware.

3 *Fundamental Concepts*

This chapter is concerned with the basic principles that underlie the 2900 design. They are mainly concerned with how programs in practice work: how they are conceived by the programmer, transformed into electrical or electronic patterns in the hardware, how execution is then carried out and how this maps back on to the original algorithm. It is in the light of these concepts that the basic architecture of the 2900 Series can best be understood.

In examining these aspects of program structure we are concerned with two interrelated ideas. First, how does the program actually use the hardware resources of store, peripherals and processors? Second, what is the relationship between the program in the programmer's mind and the actual hardware used and the software that is executed? The second idea itself consists of two parts. We need to study both the way that the final, executable representation of the program is structured and the translation process that takes place from the conceptual to the actual. Consideration of these ideas in as abstract a fashion as possible without the constraints of existing hardware and software lead to some fundamental architectural decisions. These decisions are introduced in this chapter and expanded in more detail in chapter 4.

3.1 THE VIRTUAL MACHINE

The way in which programs use store, processors and peripherals was perhaps first examined systematically in the joint development of Atlas by Manchester University and Ferranti.[4] The problems have not changed much since then. Essentially we have programs that wish to use more storage or more peripherals than are available to them, or even than there is available on the system as a whole. And conversely we have the fact that no individual program can efficiently use all the available resources all the time.

The second problem was solved by what was called at the time 'time-sharing', but is now a generally accepted technique under the name of multiprogramming. Essentially, a supervisory system shares the available processor time between several programs in order to maximise the use of other resources—peripherals, file devices, store. Individual programs, however, are unaware of this process and believe that they have the central processor to themselves—a *virtual processor*, in fact. The solution to the peripheral problem appeared long before Atlas, in IBM machines among others. If a program needs two line printers and the configuration has only one we accumulate the data being sent to the two distinct outputs on two print files and then print them sequentially on the single printer. As far as the program is concerned the effect is as if there were two printers. We have created a *virtual peripheral.* The Atlas supervisor carried this further by allowing not only more virtual instances of an available device but also the mapping of one device on to a device of a different type. Thus a program could believe it was reading cards and actually be getting data in 80-character blocks without line feeds, although the input was in fact coming from variable-length records on paper tape.

The degree of virtuality need not stop there, however. Consider access to data on a disc file. The lowest level of physical access is by addressing the disc drive via the controller. At a slightly higher level of abstraction we can address merely the drive, leaving the routing via a controller to be done by some hardware or software 'system'. Higher still we might access a named disc whose drive and controller are discovered by the 'system' but whose volume geometry we know. The next level might be to allow the 'system' to handle volume geometry and deal only with blocks, knowing their physical format and size. Or we could ignore blocking and deal with logical records. Our 'system' has now become some form of data management system. Finally, we could deal only in named items; we are supported by something like a data base system.

Each of these forms of access may be considered as a level of virtuality. Within a total system there will be programs that want to use each of these levels. Each of these degrees of virtuality should therefore be permitted. (For those concerned with efficiency, the existence of these 'layers' on the basic hardware access mechanism does not necessarily mean that top-level accesses need go through all the intermediate transformations to obtain data. There is no reason why the mapping should not be direct.)

Storage is more complex. Although prices and speeds have changed considerably since Atlas, one can still buy either fast main store or slow cheap(er) store and the price differential has not altered significantly. Programs in any reasonable period of time tend to use only a small part of their total instructions and data and hence, if we can arrange that each such part is brought into fast main store when needed, we can keep the rest on slow cheap backing store. Thus, with some degradation in speed, it is possible to run programs that are larger than the main store available; they have a *virtual store.* Again this mechanism is not even remotely new. Individual programs have used overlay techniques for years. However, program overlaying is costly since the effort of overlaying is duplicated in each program, and if the system is to multiprogram, over-all system efficiency will fall considerably if each program is allowed a partition the size of its maximum overlays.

If, however, we introduce a total-system overlay mechanism we run into other problems. Variable-length overlays in store will, as they are used and deleted, leave variable-length 'holes' of available space. This will lead to the situation where enough free store may be available to load a needed overlay but it is distributed in unusable pieces throughout the store–store fragmentation in fact. There is no complete solution to this problem but techniques that alleviate it are known. An example of such a technique is *paging*–splitting the real store into fixed-size blocks and mapping on to them, by hardware, variable-length program sections with contiguous *virtual addresses.* Individual pages may reside in secondary store until needed and then be read into any free block in main store. The actual techniques adopted in any particular 2900 system design is an implementation detail but the provision of a virtual store for each program with its own virtual addresses is a fundamental part of the 2900 architecture.

By considering these resource-mapping problems together we can see that the individual solutions that have been adopted for each set of problems actually form part of a coherent pattern. Each program can have a complete *virtual machine* with its own store, addresses, processor, peripherals and files and these will be mapped on to the real available resources by the system. It is this total concept that is perhaps the most important single item in the 2900 architecture, and which allows many of the other features to be included. While the implementation of the various aspects of the virtual machine may be far from new, the generality of the over-all concept allows for some of the 2900 design innovations.

3.2 THE PROCESS

The first step in using a computer to solve a problem is for a programmer to find or invent an algorithm. This algorithm is then converted into statements in some machine-translatable language. With knowledge of the requirements of chapter 2 we can assume that this is a high level language of some sort. It is therefore tolerably close to the programmer's initial conception and this statement is the 'highest' level of expression of the program that we need to consider. This representation is then translated by a compiler into one or more sections of machine-code representation. Some of this translation will be direct in the sense that there will be instructions that correspond in a straightforward manner. In addition some services (such as type-transformation code) that do not correspond directly to the high level language may be added by the compiler. In other words sections of code have been added to the original statement to make the algorithm work at this lower level of abstraction.

The next stage is to assemble and load the program. Unless the program is very unusual this will result in more code being added to satisfy library calls. However, even this loaded manifestation is not a complete statement of the algorithm in terms of the hardware since the program will during execution also use code in the operating system. Such use may be explicit, as in a 'get-time' command or an I/O initiation, or implicit in the allocation of resources to allow the algorithm to be executed. This conventional software model is shown in figure 3.1. The operating system code is as much part of the program representation as the square root or sort routine but it is not 'added' to the program in the same way as the other. Communication between the OS code and the rest is slow and inefficient since it involves crossing the OS–object program boundary. This is, to say the least, unfortunate since such communication can be a fundamental part of the execution of the algorithm. Logically, there is no reason why this code should not be added in exactly the same way as library routines and this, as we shall see, has many other advantages.

Applications programs
Compilers
Utilities
Library procedures
and subroutines

O.S.

Interfaces

Operating system programs
– Schedulers
– Spoolers
– Data management
– Device management
– Processor management
– Store management

Figure 3.1 Two-level model of software

The 2900 architecture therefore provides a *single virtual store* for each program, which contains *all* the code necessary to execute the algorithm. We can call the code for this total representation a *process image*. Although this provides a much cleaner solution than the conventional operating split, two other problems immediately suggest themselves. First, we cannot afford a physical copy of the operating system for each process image and we must therefore provide a means of sharing one copy of the code between several processes. This mechanism will not only apply to the operating system procedures but to library routines and complete utilities. It can allow, for example, multi-access users to share a copy of compiler. We can also use it to share user-produced code between co-operating programs.

The other advantage of the two-level model is that software below the line of figure 3.1 is protected from errors in software above it by the needle eye of the OS interfaces. This, however, is a very crude mechanism since the single 'fire wall' merely divides the code in a fairly arbitrary manner and does not prevent errors in, say, a spooler from propagating through the complete operating system without check. And one pays a considerable overhead penalty each time the fire wall is breached. We must, therefore, also provide some more general and more efficient mechanism for protecting both code and data. Such a mechanism can allow for multiple levels of protection depending on the *privilege* of the code involved. Such a possible multilevel organisation is shown in figure 3.2.

This provides an additional advantage. Some of the routines (such as conversion procedures) below the OS-interface barrier of figure 3.1 would be of use to an applications program but are not used because of the high overhead of an OS call. Indeed, in general there will be no OS call that provides access to them. With a multilevel approach and lower overhead protection mechanism such facilities can be used, avoiding duplication of storage and effort. This in effect provides the basic mechanism for the user to select the degree of virtuality he requires in peripheral access. The different data-access methods of the last section can be thought of as procedures within the system software, which call on each

APPLICATION PROGRAMS
e.g. COBOL program for information retrieval

CONTINGENCY AND SERVICE PROCEDURES
e.g. recovery of file after system failure

JOB COMMAND LANGUAGE INTERPRETER
for control of program and system

RECORD MANAGEMENT
blocking/unblocking files –
serial, indexed sequential, etc.

CATALOGUE MANAGEMENT
user, file, volume management services, etc.

PHYSICAL DEVICE HANDLING –
card read/punch, discs, operator's console, etc.

PROCESSOR AND DEVICE DISPATCHING
system error handling
store management

HARDWARE

Figure 3.2 Multi-level model of software

other for services. Choosing the degree of virtuality is then equivalent to choosing the particular procedure within the chain at which the application program is to enter. Again, because there is now no logical difference between routines in the operating system and in applications libraries, providing that the protection system is designed in the right way, the access to a procedure at any level of privilege can be identical–specifically a standard procedure call. This means that a compiler need have no knowledge of whether an external facility is provided by a library procedure or the OS (deciding whether to generate a call or an OS macro is always messy). More important even, the compiler need have no knowledge of whether the code it is compiling is to be *part* of the operating system or not. The way in which it invokes other OS facilities will be identical to the way in which they are called by an unprivileged applications program. We do not therefore need special compiler versions (or worse, languages) for operating system development.

3.3 PROGRAM STRUCTURE

The final set of fundamental concepts arises from consideration of the structure of programs in their high level language form and attempting to find ways to provide a system on to which this natural structure can be easily mapped. What then are real programs actually like? Examination shows that most are often badly and illogically structured, but that modern programming techniques are attempting to eliminate this. The basic unit of structure tends to be a subroutine, procedure or *module*; that is, a set of code for carrying out a well-defined subset of the algorithm with its own constants, named scalars and data structures. The data structures could be simple things like an individual array or a complex entity such as a FORTRAN COMMON block.

Figure 3.3 shows the structure of a very simple two-module process, using the terms discussed in section 2.3. The two modules each have their own code

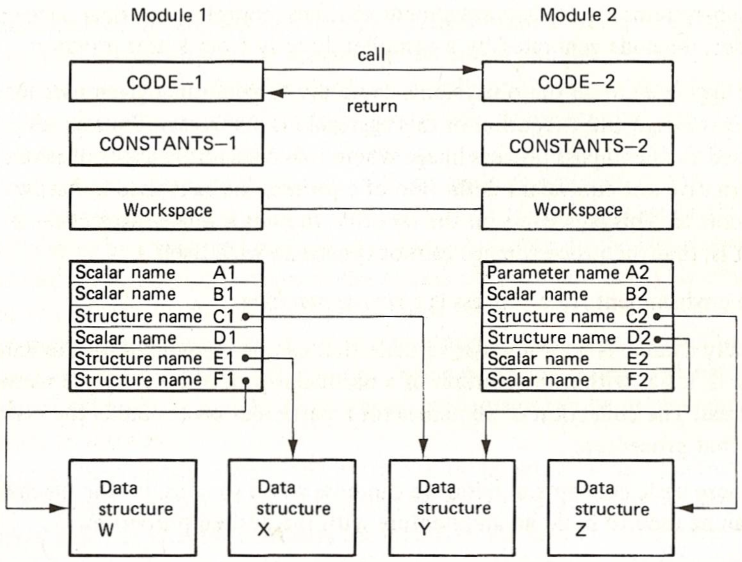


Figure 3.3 Logical process structure

and constants and a set of named items. Some of these latter are scalar variables, others are the names of structures that are represented separately. In addition, there is a need for general work space for use during the process execution. The size of this area will, unlike other areas, vary dynamically as the modules are executed. In the diagram Module 1 calls Module 2 as a procedure and passes to it a scalar parameter, which is referenced in Module 2 as the first of its named items, A2. Note also that the two modules share access to Data Structure Y. This is common since passing complete data structures as parameters is time-consuming. Sharing scalars is less common but possible, and they can be considered as degenerate structures. In addition, in some block-structured languages at least, it may be possible to access the local names of Module 1 from Module 2.

What we require in an architecture, then, is a natural support for this form of program structure, and if we are to allow languages such as Algol or PL/I we must also have system (and preferably hardware) support for recursion in individual procedure modules. Note that this form of logical structure leads naturally to the use of easily shared, pure, code since CODE-1 refers to such locations as NAMEA1 and, by providing a new block of names, the same code can be used without interference.

We are now in a position to define more formally some terms that have already been rather loosely used in earlier sections.

- We will call a module with the structure shown in figure 3.3 a *procedure* or *procedure module.*
- Such procedures can be grouped into aggregates that perform together some coherent and well-defined function, and present to other aggregates a coherent and well-defined interface. An aggregate of this kind is a *subsystem.* Examples of sub-systems are: data-management routines; compiler run-time packages; a loader; the code generated by a compiler directly from a user program.
- The aggregate of all the subsystems necessary to perform a given user task is a *process image* and execution of this aggregate is a *process*. (Sometimes process is used to encompass process image where this does not cause confusion. An alternative but equivalent definition of a process image is used in hardware definition. This lays stress on the fact that there is a single instruction stream, that is, that the process image cannot timeshare with itself.)
- The environment for a process is a *virtual machine.*
- Finally a *name* is a local scalar variable that can be recognised by the hardware, that is, it has within the context of a particular procedure a unique name or address. The collection of all names for a particular procedure is the *namespace* for that procedure.

With these basic concepts in mind we can now go on to consider the mechanisms that can be used to build an architecture with the desired properties.

4 *Architectural Mechanisms*

One can think of the architecture of a computer system as being a mixture of the most important and interesting features of the design and the relationship between these various aspects. In defining the general architecture for the 2900 Series the feasibility of implementation had to be continually borne in mind, but the basic concepts were kept simple and uncluttered by the actual methods of building such a system, except where consideration of these methods was vital to the achievement of the architectural designers' aims. Any specific 2900 system design would have to be a practical realisation of these concepts. In the last chapter we considered the structural features of the system that we required. This chapter considers the basic mechanisms by which these features could be provided. These mechanisms are still described conceptually, actual implementation details for hardware or software being included only when necessary to aid explanation. Chapter 5 will go on to consider how these mechanisms can be implemented in hardware or software.

To map these divisions on to the more traditional world of architecture, chapter 3 considered what sort of building we were to produce in the light of predicted use; this chapter considers the layout of the building, the room dimensions and services required; and chapter 5 considers the actual building production, the plumbing, electricity supply and building standards.

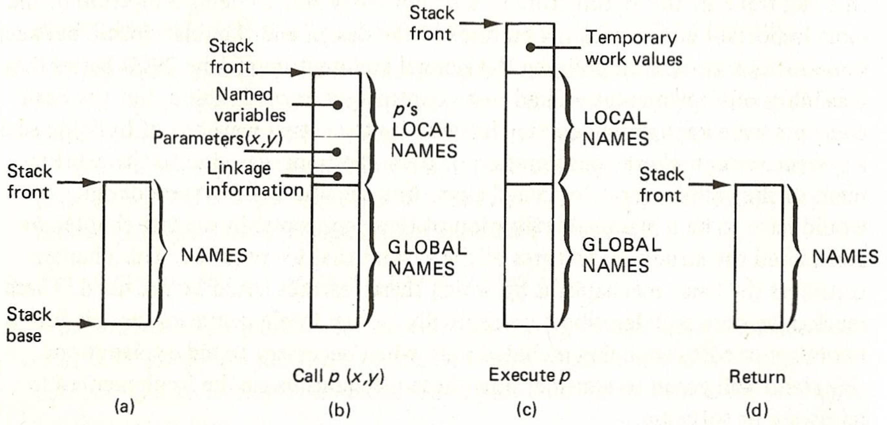
In defining the mechanisms it is easier to move from the internal needs of a particular process outwards to the total system, and so we begin with the methods of realisation of the process layout described by figure 3.3. The software terminology used is that of VME/B (see section 6.3).

4.1 THE PROCEDURAL STACK

The basic mechanism for the implementation of procedural storage requirements is a hardware-managed stack. A *procedural stack* is a set of consecutive storage locations to which individual data items or blocks of data can be added and from which they can be removed in a last-in/first-out fashion. Each virtual machine, and therefore each process, has its own stack. Whenever a procedure is called, namespace for it is allocated on the stack. As the code of the procedure is executed any temporary work space required is also allocated and removed when no longer needed. When the execution of the procedure is complete the namespace is also removed, returning the stack to its state at the time of the call.

The procedural stack can best be imagined as having a vertical structure with items being added to, or removed from, the top of the structure (see figure 4.1a). However, items within the procedural stack can be accessed without removal so that, for example, a procedure can access global data declared in an enclosing procedure. The procedural stack is therefore not a 'pure' hardware stack of the kind found in English Electric's KDF9 although, as we shall see, the top of the procedural stack exhibits the same properties.

Figure 4.1a shows the state of the stack at the moment before the call of a procedure *p* that has two parameters *x*, *y*. The form of this procedure is shown



```
Procedure p (x,y)
declare variables a,b, . . .
a = x + 1
. . .
. . .
return
```

Figure 4.1 Stack processing

at the bottom of the figure in an undefined but hopefully self-evident language. At the moment of call (figure 4.1b)\ linkage information is placed on the stack in preparation for the eventual return from *p*. The stack front is then raised to allocate space for the parameters and locally declared names of *p* and the actual values of *x* and *y* are placed on the stack. Procedure *p* is entered. It has all of its local names to hand and if it uses any names of surrounding procedures these are available to it in the global namespace on the stack.

As *p* is executed (figure 4.1c) it may need temporary workspace, for example to store partial results of expression evaluation. These will be allocated above the local names by raising the stack front to accomodate them. As they are used, and no longer required, the stack front automatically falls, giving up the unwanted space. This section of the stack, then, acts as a pure last-in/first-out stack for the purpose of expression evaluation. Should *p* itself call another procedure in the middle of such an evaluation the namespace of the new procedure would begin above the current stack-front level, thus automatically preserving any temporary values until return.

When *p* has completed execution (figure 4.1d) control is returned to its caller via the linkage information on the stack and the stack front is returned to its position immediately before the call of *p*, thereby freeing *p*'s namespace for further use.

Figure 4.2 shows how the procedural stack mechanism can be used to map the namespace of the procedures of figure 3.3. The illustration is of a point during the execution of module 2, which has been called from module 1. Essentially

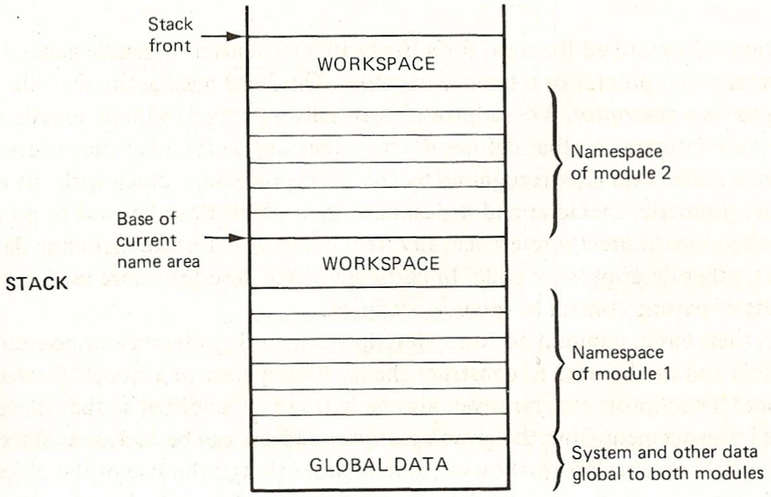


Figure 4.2 Process mapping–stack

the name spaces of the two procedures have been mapped without change on to the stack while the work space is provided automatically and dynamically at the top of the stack. The layout illustrated assumes that module 1 has some temporary values that it wishes to retain between the call of module 2 and its return (for example, it is calling module 2 as a function in the middle of an expression evaluation). Were it not for this fact the work space above the namespace of module 1 would not exist. The stack also contains global data that was not declared by either module. This is assumed to be the namespace of other subsystems in the total process–for example, the operating system. Note that logically the data structures themselves could also have been placed on the stack (the shared structure B would have to be in the namespace of module 1 to allow access by both modules). There is nothing in the architecture to prevent this and, indeed, small structures do sometimes reside on the stack in a 2900. However, general properties of structures and access-efficiency considerations, as we shall see, make it more useful to segregate and store them separately in most cases, with pointers to them from the stack.

Finally, for any reader concerned about the temporary existence of procedural names for such static languages as FORTRAN, although the official FORTRAN definition does permit a dynamic stack implementation, there is nothing to forbid the concatenation of names from several FORTRAN subprograms at load time in order to provide essentially a static environment for the FORTRAN-coded section of the total process. The system software that lies below this section, and runs the procedures called in by the program, may still utilise the dynamic nature of the stack.

4.2 DESCRIPTORS

Section 2.3 described the need for a hardware mechanism to handle general data structures and pointer or reference variables. The 2900 mechanism for this purpose is a *descriptor*. A descriptor is essentially a (virtual) address together with reference information that defines the item that can be found at that address. It forms a scalar data type recognised by the 2900 processors, along with the more usual arithmetic, character and logical data. A descriptor can be used to point to, and therefore indirectly reference, any item in the virtual store including data items, other descriptors or code. In particular, code descriptors are used as the means of passing control to other procedures.

In their basic, unmodified form, descriptors map the reference or pointer variable and can be used to construct chains, lists or trees in a straightforward manner. Descriptors can, however, also be indexed or modified so that items at a stated displacement from the given descriptor address can be accessed. Since the descriptor contains information on, among other things, the size of the object pointed to, the displacement can be stated in terms of items and the hardware can automatically scale this to provide the correct virtual address. This provides an efficient access method for structures of the form of arrays or strings. Finally the positions of the index and address can be reversed to give reverse modification. In other words it is possible to construct a descriptor that references an item of a particular type that is not at a particular virtual address but at a stated displacement from an unknown address. If an address is now provided in the form of a modifier the descriptor will reference the item of the stated type at the stated displacement from the address. This mechanism can be used to map a particular data structure on to a given unstructured area of store and thus forms the basis for 2900 structure handling.

The value of the descriptor concept was originally demonstrated in the Basic Language Machine (BLM).[1] This proved the usefulness of adopting an architecture in which information structure could be explicitly described within the machine rather than working within a fixed reference frame dictated by the actual mechanics of the storage mechanism. While, because of conflicting requirements, the Synthetic Option found it impossible to carry this philosophy to the limits set by the BLM, Iliffe's work did have a considerable effect on 2900 thinking.

Using descriptors, we can extend the representation of figure 4.2 to include the data structures and show that access to the structure elements will be by descriptors held in the stack namespace. This is shown in figure 4.3. Note that the call between module 1 and module 2 will also be carried out via a descriptor and that the linkage information held on the stack for return to module 1 will effectively be in descriptor form.

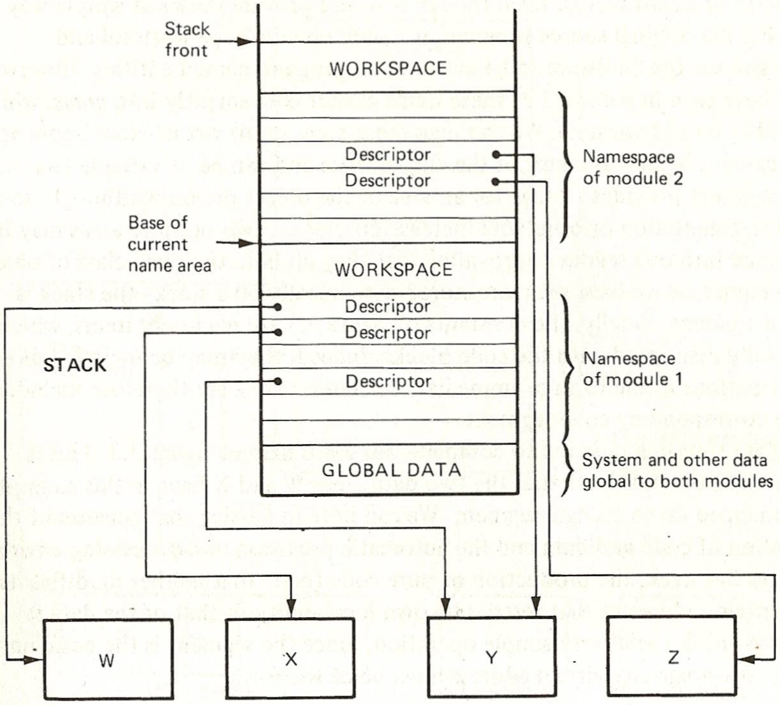


Figure 4.3 Process mapping–descriptors

4.3 THE VIRTUAL STORE

As we have already seen the virtual machine is the environment for a single process; there is a one-to-one correspondence between active processes and virtual machines. We can best study the form of the virtual machine in the light of the conceptual requirements of chapter 3. Section 3.1 required that the virtual store concealed from the individual programmer the knowledge of the storage hierarchy of the system. For him there must be only a one-level storage, with any automatic overlaying being done behind the scenes and out of his sight. Second, within this store will reside a collection of subsystems. For an individual applications programmer some of these will be known–he will have written them or directly included them; some will be implied–he will have invoked library functions; and others will be unknown–they will have been included by the compiler or loader for their own purposes or called indirectly by routines that he directly requested. However, the form of the individual subsystems will be identical to his own subsystem; to their authors they will have exactly the same structure as his own code. He will refer to all known data and code by a combination of name (for example, a procedure, label or structure name), displacement (an array index or character number within string), and length (for example, an 8-bit character or 32-bit word). Even though it would provide the most simple way of mapping the original source program, it would obviously be wasteful and expensive for the hardware to be aware of all program-named entities. However, as we have seen in section 3.3, these items cluster conveniently into *areas*, which can either be code or data. We can map these areas on to virtual-store *segments*. The segment is the basic unit of the virtual store and can be of variable length. Each segment provides storage for an area of the object program although, to avoid fragmentation or otherwise increase efficiency, two or more areas may be combined into one segment, providing that they all hold the same class of object. Local names, as we have seen, are stored dynamically on a stack–the stack is itself a segment. Finally, the constants of figure 3.3 are read-only items, which are totally associated with the code blocks. Indeed, they may be included directly in instructions in the form of immediate operands. They are therefore included in the corresponding code segment.

We are thus in a position to complete our 2900 map of figure 3.3. This is shown in figure 4.4. Note that the two data areas W and X have in this example been mapped on to a single segment. We can note in passing that because of the separation of code and data and the automatic provision of a processing environment by the stack, the production of pure code (code that neither modifies itself, nor contains addresses that restrict its own locatability or that of the data it operates on) is a relatively simple operation. Since the segment is the basic unit of virtual storage any virtual address must be of the form

N	D
Segment number	Displacement

This virtual address is mapped on to the real store address via a *process segment table.* The *n*th table entry corresponds to segment *n*. An entry in the segment table has the following form

P	R	L
'Present' marker	Real start address	Segment length

The 'present' marker *p* indicates if the segment is currently in main store or not. If not, when an access attempt is made, a *virtual store interrupt* is caused, which will bring in the segment from backing store. This is the basis of the one-level store.

Another requirement of the virtual store was that it should be possible to share code or data between processes. This could be achieved by mapping the same real address on to segments in different processes, that is, the same *r* appears

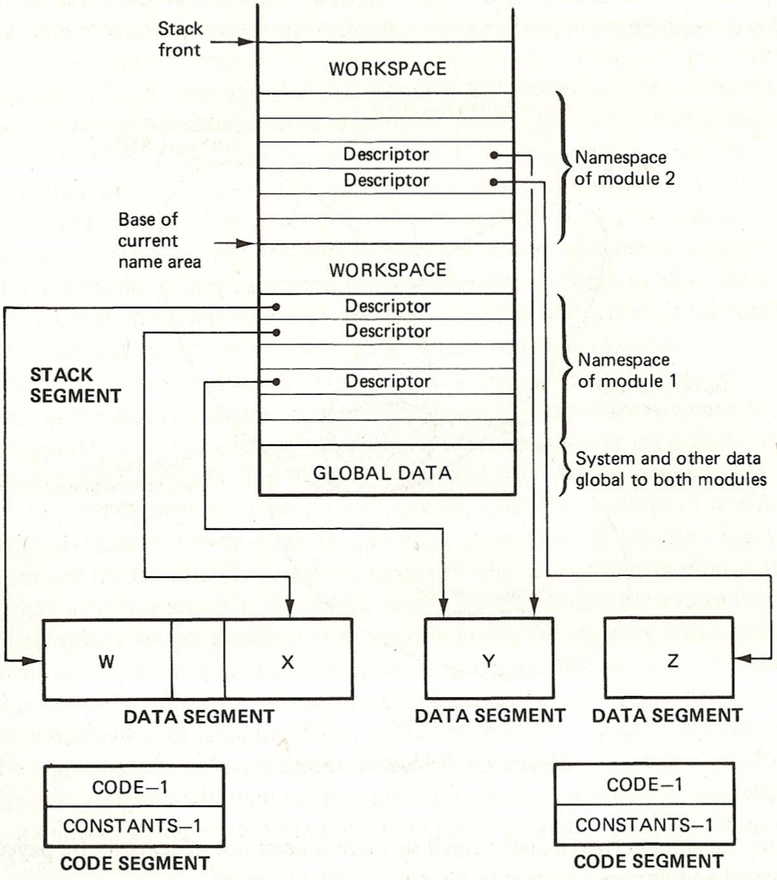


Figure 4.4 Process mapping–complete

in more than one place. The two processes can thus refer to the common segment by different virtual segment numbers. (While this is conceptually the method by which interprocess sharing is achieved, in practice–to avoid system housekeeping problems caused by the existence of multiple copies of real addresses–another level of indirection is employed. Since this is more concerned with implementation than mechanisms it is dealt with in the next chapter.) This basic mechanism provides the means of sharing essential supervisor code between all processes, as well as more 'casual' sharing between two or more communicating processes. Figure 4.5 shows schematically this overlap of virtual store. Note that, since we are talking of virtual store only, any of the segments of figure 4.5 whether shared or not could at any instant in time be in main store or in any other part of the storage hierarchy.

As we have seen in section 3.1, because segments are of variable length, store-fragmentation problems may arise. As an implementation choice, therefore, rather than a fundamental architectural requirement, individual 2900 implementations employ a fixed-length paging scheme to support the segmentation scheme.

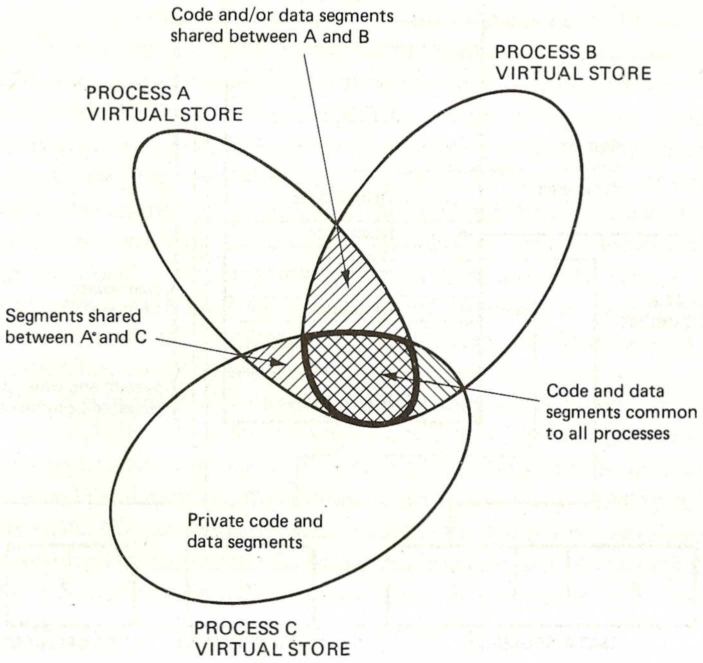

Code and/or data segments shared between A and B
PROCESS B VIRTUAL STORE
PROCESS A VIRTUAL STORE
Segments shared between A and C
Code and data segments common to all processes
Private code and data segments
PROCESS C VIRTUAL STORE

Figure 4.5 Virtual store sharing–1

However, segments (particularly small segments) need not necessarily be paged, and paged and unpaged segments can co-exist in a process.

Finally, since segments are addressable entities and each contains information of a single coherent type, they form the natural units for protection purposes as part of the protection and privilege scheme. This is discussed in section 4.4 below.

The advantages of the form of segmentation adopted for 2900 can thus be summed up as follows

- direct mapping of program structure
- concealment of multi-level storage hierarchy
- basis for pure code generation
- provision for code and data sharing between processes
- basis for protection scheme
- ability to be underpinned by other mapping techniques (such as paging) while not actually demanding such assistance.

4.4 THE PROTECTION SYSTEM

A minimum requirement of the process concept was that the 'operating system' should be protected from the user process. However, we are looking for a much more comprehensive form of protection than that, as section 3.2 explained.

One aspect of protection is already implied by virtual addressing. Unshared segments cannot be corrupted (or even accessed) by other processes since they cannot use real addresses and they will not have any virtual address corresponding to the real address of the unshared segment. A second protection aspect can be easily deduced from the definition of a segment. Since a segment contains only data or code items with common characteristics, these characteristics can be recorded and monitored by hardware on every access. Thus a code segment can be marked as 'execute only' and can be protected from accidental writing or reading within the process. Data segments can be marked both as 'data' (to prevent accidental execution) and as either 'read only' or 'read and write'. Vital constants can thus be protected from accidental corruption within the process.

However this is not sufficient. One can easily imagine data—peripheral allocation tables, for example—which must be updated by a system procedure, but which should be protected from being overwritten by all other procedures. Again, the loader subsystem will read and write data that will then become executable code. To cope with these situations we must introduce the concept of *privilege.* A classical two-level OS-application split has a crude form of privilege. Everything below the OS-interface level has more privilege to read tables, alter data, etc., than anything above it. We need a finer graduation than this and so the 2900 provides multiple *privilege* or *protection levels.* We can now assign each segment in a virtual machine a separate *protection key* (that is, read, read and write, execute) within each privilege level. Thus the most privileged procedures, those concerned with mapping virtual resources to real resources, will find most of the segments of a virtual machine available for writing, while a user-produced application program will tend to find that only its own names are so available. To paraphrase Orwell, within the process all subsystems are equal, but some are more equal than others.

All that is needed now to complete the protection system is a means of identifying the point at which a privilege boundary is crossed. We need to provide this without relying on the programmer's or compiler's knowledge of what privilege applies to which subsystem. This would involve too much system knowledge, increase the chances of error and complicate such operations as running the same procedures at different levels of privilege for different purposes. The basis of a suitable mechanism is already available, since we have noted that the only way of accessing code in another subsystem is by procedure call and that calls are made via code descriptors. The call instruction can remain the same regardless of whether the procedure called has the same or a different privilege. The relative privilege levels will be known by the loading subsystem, which can therefore amend the descriptor accordingly (see figure 4.6). When the call is executed, if no privilege change is involved a normal procedure call is carried out.

If there is a change, the descriptor access will cause what is known as a *system call* to occur, and the necessary changes to access keys and provision for re-setting them on return from the procedure can be made.

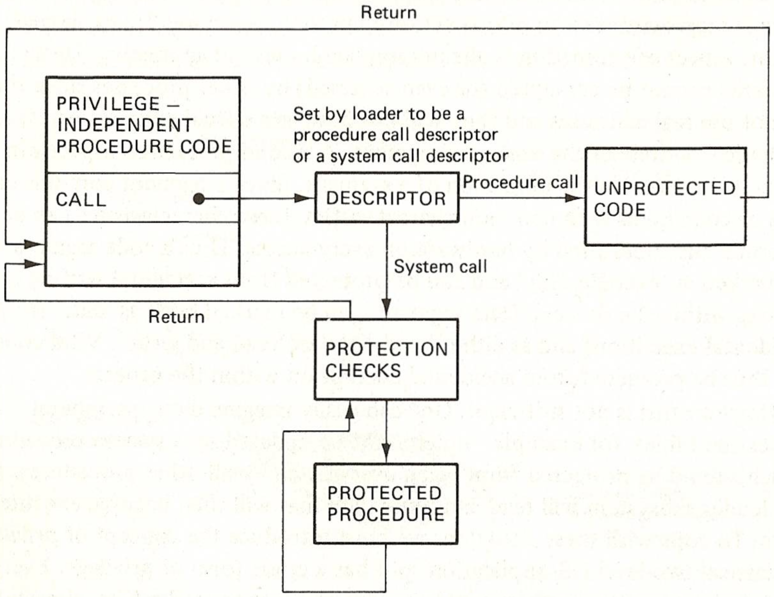

Return
PRIVILEGE – INDEPENDENT PROCEDURE CODE
CALL
Set by loader to be a procedure call descriptor or a system call descriptor
DESCRIPTOR
Procedure call
UNPROTECTED CODE
System call
Return
PROTECTION CHECKS
PROTECTED PROCEDURE

Figure 4.6 Procedure and system calls

This mechanism involves only limited software knowledge (in fact, it is confined to the loader), can be easily supported in hardware, and provides protection checks on every store access without any time penalty since the check can be in parallel with other operations. The way in which the protection system is implemented is described in detail in section 5.4, and so the current description of its properties is kept deliberately brief. Readers confused by unfamiliar concepts or unconvinced that the system contains no loopholes should suspend judgement until chapter 5.

4.5 PROCESS CONTROL

A total 2900 system is a collection of virtual machines and all system procedures needed are in every virtual machine. In such a set-up we need to solve the problems of inter-process control. Two questions immediately spring to mind—what creates virtual machines and what allocates real resources to them? The question of how processors are assigned to particular processes is especially intriguing since the scheduler is itself within the process. How then do we deal with suspended processes and interrupts? It is obvious that for the system to work, virtual machines must co-operate in some way, but how are they forced to?

The answer is a subsystem of the highest privilege which deals with real resources. This subsystem is called the *kernel* and corresponds to the layer of software immediately above the hardware in figure 3.2. Its chief functions are to allocate real resources, such as peripherals or store, to virtual machines; to handle system errors; and to deal with interrupts. The kernel exists in all processes and is known by the hardware. Whenever there is an interrupt or failure the kernel is entered by the hardware after the current process state has been recorded. To preserve the general rule that all code access is by procedure call, the hardware can invoke the kernel by a forced procedure call, giving it the necessary information on the interrupt or fault as parameters. There is only one difference from a normal procedure call, which is that the kernel must have available to it real work space on entry (otherwise there will be a virtual store interrupt, which will invoke the kernel, which will not have work space, which will cause a virtual store interrupt, . . .) The section of the kernel that deals with faults and interrupts is called the *event manager*. Its existence solves the problems caused by having the scheduler within the process and can also provide the basis for an interprocess communication system. One process can activate or otherwise communicate with another by making operative a *named event*. This can be treated by the event manager as a form of interrupt, which it processes using *event tables* for synchronisation, etc.

The virtual-machine system means that all internal system-procedure interfaces can be visible to application-program modules. This is illustrated in

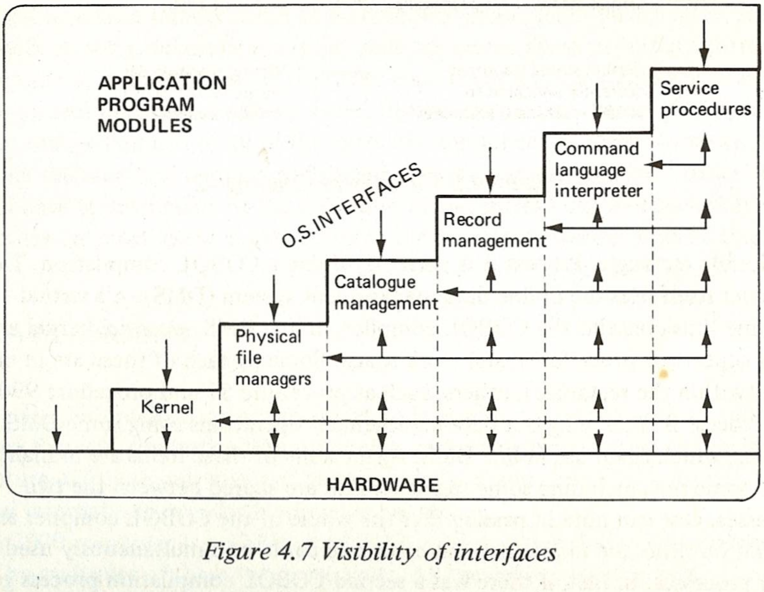


Figure 4.7 Visibility of interfaces

figure 4.7, which shows that although the system is layered as in figure 3.2, any 'layer' can call for services (represented by the arrows) from any other layer, and application modules can interface directly with any layer in the system.

Figure 4.8 reillustrates the segment-sharing concepts of figure 4.5 for two virtual machines but adds to it actual examples of shared subsystems, and maps the combined processes on to real resources. Real resources are shown within

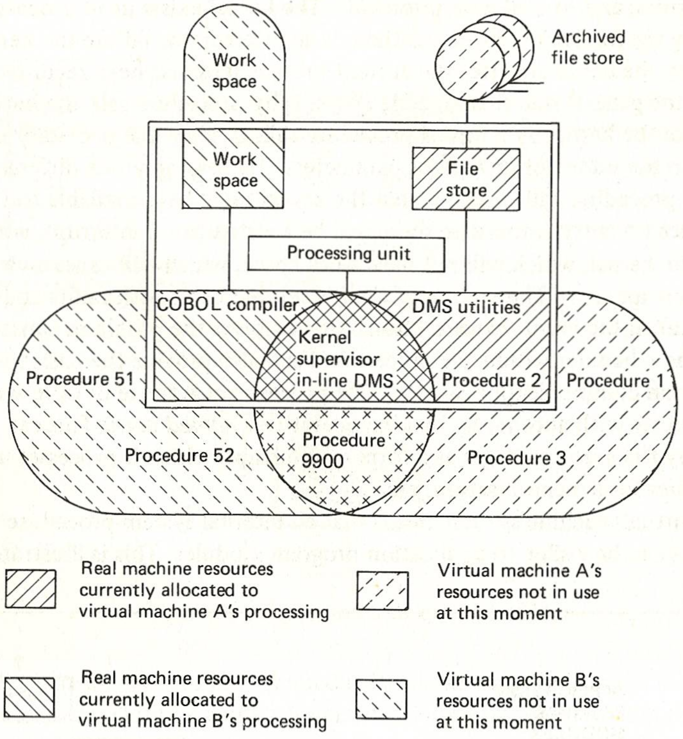

Work space
Archived file store
Work space
File store
Processing unit
COBOL compiler
DMS utilities
Kernel supervisor in-line DMS
Procedure 51
Procedure 2
Procedure 1
Procedure 52
Procedure 9900
Procedure 3
Real machine resources currently allocated to virtual machine A's processing
Virtual machine A's resources not in use at this moment
Real machine resources currently allocated to virtual machine B's processing
Virtual machine B's resources not in use at this moment

Figure 4.8 Virtual store sharing–2

the double rectangle. Process A is currently doing a COBOL compilation. The compiler itself uses the in-line data management system (DMS). A's virtual machine thus contains the COBOL compiler, in-line DMS, essential kernel and other supervisor procedures, and work space. Some of each of these are in main store (within the rectangle); others, such as procedure 51 and procedure 9900, are not. Process B is carrying out data-management operations using some DMS utilities, which again use in-line DMS. Again some of these items are in main store, some not, including some segments that are shared between the two processes. One can note in passing that the whole of the COBOL compiler and the DMS utilities are in practice shareable and could be simultaneously used by other processes. In fact, if there was a second COBOL compilation process going on in parallel, the whole of the contents of virtual machine A would appear in the second virtual machine, except for work space and data areas used to hold source, object and intermediate code associated with each particular compilation.

5 *The Primitive Architecture*

The primitive architecture is the way in which the fundamental concepts and mechanisms of the two previous chapters are implemented in actual systems. Some of these implementation methods are a fundamental part of the 2900 Series, but most are merely choices for one or more of the members of the Series that were implemented first. They thus form subrange standards or particular machine implementation details. Only those that are of importance in understanding 2900 concepts are included in this chapter and so the description is not a complete one for any particular range member. For example, error-checking mechanisms are a vital part of any particular implementation but they are omitted from the description except where they throw light on basic architectural matters.

Conversely, one cannot assume that all future members of the 2900 Series will exhibit all the standards discussed here. For example, in this chapter, and in appendix A, some details of the order code are given. These details are correct at the time of writing but since the order code is *not* a range standard, future 2900 systems may have a different one. Indeed, there is no reason why it should not be changed on one or all of the currently announced models. However, because of other architectural features and design aims, any new or amended order code would have to have many of the attributes of the current one and such abstract attributes are most easily explained using the current definition as an example.

5.1 DATA FORMATS

The primitive architecture at the object code level is concerned with such items as data formats, the instruction repertoire, addressing mechanisms and register structure. Of these only the first are range-defined and conform to formal or actual international standards wherever possible.

A 2900 computer is a 32-bit-word, 8-bit-byte machine. Individual data items may be multiples of these two basic units. All the data types required by common high-level languages are provided

- fixed-point (integer)
- floating-point (real)

- decimal (binary-coded decimal representation)
- logical (bit-pattern or unsigned integer)
- character (binary-coded character information).

Fixed-point numbers are held in two's-complement form and can be one or two words long. Floating-point numbers are held with a hexadecimal exponent and may occupy one, two or four words. Decimal numbers have four bits allocated for a sign and four further bits for each decimal digit. In store they can occupy any integral number of consecutive bytes up to a maximum of 16, that is, they can have a sign plus any odd number of digits from 1 to 31. However, for arithmetic purposes they are always expanded to 7, 15 or 31 digits corresponding to one, two or four words. Logical data items may be one or two words. Finally a character occupies one byte. As we have already seen, there is one other data type fundamental to the architecture—the descriptor. A descriptor always occupies two words, of which the second is a virtual byte address. The first or most significant word defines the type of the descriptor and gives information about the item addressed. There are four basic types of descriptor.

(1) *Vector descriptors* point to a vector or contiguous list of items in store. The first word of the descriptor states the size of the item accessed, which may be one bit, one byte, or one, two or four words. It also states the length of the vector and whether modified accesses via this descriptor should be hardware-checked against this bound. Finally it states if any modifier should be scaled by the item size. (The normal case is to scale so that modification by *n* will access the *n*th list item rather than the *n*th byte.) Modification will be unscaled if, for example, the descriptor is being used to map a structure on to a store area, the address of which is used as a modifier. The second word of the descriptor would then normally contain a byte displacement.

(2) *String descriptors* are used to identify a string of bytes. They specify the address of the first byte and the length of the string.

(3) *Descriptor descriptors* can only point to other descriptors. They are similar to vector descriptors although the item size must always be two words.

(4) *Code descriptors* are used to identify code.

As described in section 4.4, a code descriptor can be either a *procedure-call descriptor*, or a *system-call descriptor* if a privilege boundary is to be crossed. Both are used in call instructions and for return linkage. There is a third subtype called an *escape descriptor*. If an escape descriptor is discovered by the hardware accessing mechanisms when they expected one of the other descriptor types, a jump is made to the escape-descriptor-specified address. The state of the machine is preserved so that the code accessed in this way can construct an actual descriptor to replace the escape descriptor and restart from the point at which the escape descriptor was discovered. This provides a very convenient and efficient way of implementing such high level language mechanisms as the full generality of Algol 60's call-by-name. It also supports run-time code binding, that is, the ability to delay linkage of little-used routines until the first time that they are actually called.

A more detailed description of 2900 data formats can be found in appendix B.

It is also possible to hold descriptors in store with their address field containing a displacement relative to the storage position of the descriptor itself. Facilities exist for automatically constructing a normal descriptor from such a self-relative descriptor.

5.2 INSTRUCTIONS AND REGISTERS

This section is totally concerned with the 2960/70/80 implementations but exhibits the general principles that need to be shown by any 2900 order code and register structure as well as the associated function of data addressing. The section does not attempt to define the complete order code or list all registers but appendix A gives more details. We have seen that all directly addressable items such as local names, parameters and work variables are held on the stack, off-stack variables being accessed via stacked descriptors. Each virtual machine has its own stack, which constitutes one of its segments. The stack is one word wide, that is, items are added to it or removed in units of one word. The register structure supporting the stack is shown in figure 5.1. The *local name base* register,

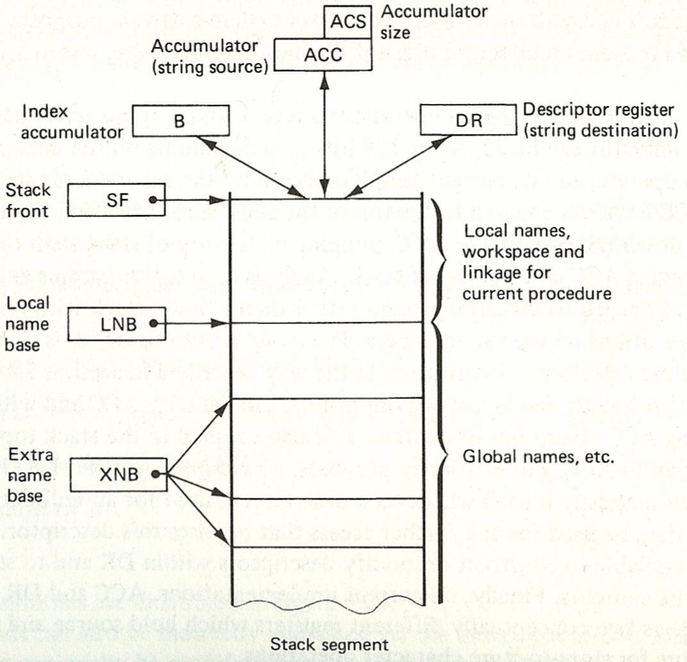

ACS
Accumulator size
Accumulator (string source)
ACC
Index accumulator
B
DR
Descriptor register (string destination)
Stack front
SF
Local names, workspace and linkage for current procedure
Local name base
LNB
Extra name base
XNB
Global names, etc.
Stack segment

Figure 5.1 Basic register structure

LNB, points at the base of the local data of the currently active procedure and local names are accessed relative to the value in LNB. The top of stack is indicated by the *stack front* pointer, SF. This controls allocation and release of new stack storage, provides a check against access of currently undefined storage above the stack top and allows addressing of the pushdown workspace at the top of the stack.

When a procedure is to be called, LNB can be made to point to the current value of SF and linkage information is dumped on the stack. Linkage information will include not only the return address but also the old value of LNB, so that the previous stack status can be restored on return. It is not necessary to save the old value of SF since this will be the new value of LNB.

Procedures in block-structured languages may also access names declared in surrounding procedures, which will be at lower levels on the stack. Such access could have been provided within the architecture by a set of display registers that would point to all previous name bases but analysis of real programs showed that this would not have been cost-effective and the facility is instead provided by a single register, the *extra name base* pointer, XNB, which can be set to any desired point within the stack. Items on the stack can, of course, also be accessed indirectly via descriptors and this method is used for passing parameters by name. Also, as an implementation decision, XNB is, unlike SF and LNB, capable of holding a full virtual word address and so it can, if necessary, be used as a base address for directly accessed off-stack areas. This is useful in non-block-structured languages. (There is actually a second register, LTB, functionally equivalent to XNB, but since it is conventionally used for certain system-software purposes outside the scope of the main architecture it is not further described here, but in section 5.4 below.)

Most arithmetic and logical operations are performed in the *accumulator* ACC. The accumulator can be 32, 64 or 128 bits long depending on the data on which it has to operate, and its current length is shown by the *accumulator size* register, ACS. ACC is shown coupled to the top of the stack since provision exists for pushing down the contents of ACC, popping up the top of stack item to ACC, or interchanging ACC and the top of stack. Analysis of actual programs and efficiency considerations led to this arrangement rather than a 'pure' stack with reverse polish operations on the top members. The *index accumulator,* B, is used mainly for dynamic descriptor modification in the way described in section 5.1 above. Modifier arithmetic can be carried out in B in parallel with ACC and without disturbing ACC operations or content. B is also coupled to the stack top in a similar way to ACC. For efficiency purposes, a *descriptor register*, DR, is provided. DR is automatically loaded whenever a descriptor is used for an indirect access and can then be used for any further access that requires this descriptor. Facilities are also available to construct or modify descriptors within DR and to stack and unstack its contents. Finally, on current implementations, ACC and DR are used to double as two conceptually different registers which hold source and destination descriptors for store-to-store character operations.

The dedication of registers to particular purposes, rather than provision of multiple general-purpose registers, allows easier exploitation by programs and permits engineers to optimise each register's design to its chosen purpose. Because of the close mapping of program structure on to hardware, any extra money that the hardware designer might have can be devoted to very efficient slaving registers, which will be automatically exploited by programs without alteration to code. For example, because most accessed items are always in the local name base, a stack slave that keeps for future immediate access the last few items obtained relative to LNB must have a very high hit rate, an order of magnitude higher than a general cache slave on a conventional machine. In addition, hardware registers slaving the top of the stack have a similar effect to multiple-accumulator provision without the need to optimise software, and without the need for software to deal with the situation that arises when an expression evaluation uses more pushdown registers than are available. Again hardware organisation and design can take advantage of the fact that code and data are always in separate segments and that off-stack segments are always addressed indirectly.

2900 order-code functions have in general two operands, with one usually in a register implied by the function code. However, store-to-store operations to move, compare, edit and manipulate character strings are also provided. The function code and addressing form are completely dissociated. This has three major advantages. First, it makes instruction generation by high level language compilers easier. Second, it allows any function to take either a 32-bit form or a short 16-bit form if the address representation will allow. Third, it permits easy implementation of hardware pipelining, where operands can be fetched for instructions while previous instructions are still in various stages of execution. Basic directly addressed operands for arithmetic and logical operations are as follows

	Operand	*Normal use*
(1)	literal item in the instruction	simple constant value
(2)	item at fixed displacement from current instruction	complex constant value
(3)	item at fixed displacement from LNB	local name
(4)	item at fixed displacement from XNB (or LTB)	global name
(5)	item at the top of stack (TOS)	partial result
(6)	contents of B	index for manipulation

These possibilities are illustrated in figure 5.2a..

Operands can also be indirectly addressed via the descriptor in DR. This descriptor can optionally be modified by an index indicated by any of the forms 1 to 6 above. Finally, an operand can also be addressed via a descriptor held in any

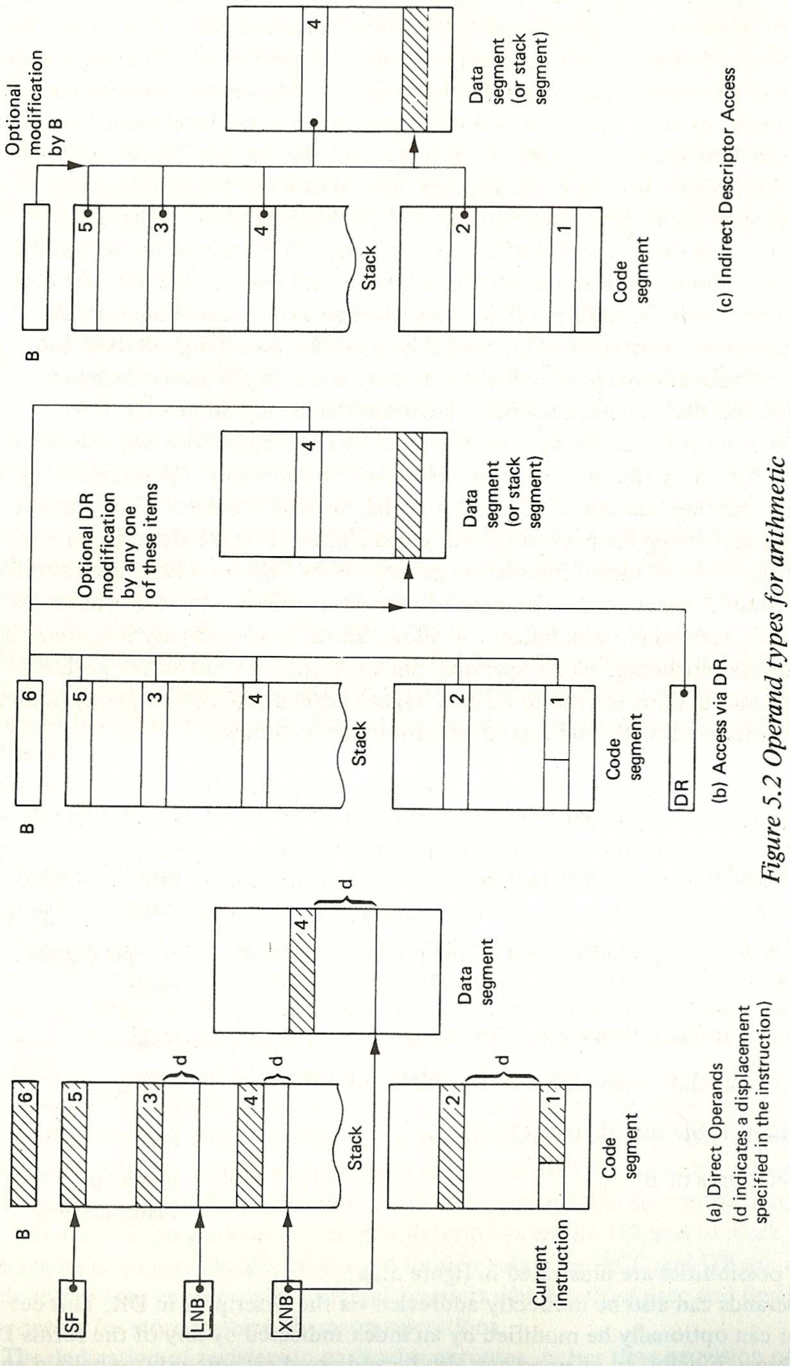

SF
LNB
XNB
B
6
5
3
4
d
Stack
2
1
Code segment
Current instruction
Data segment
(a) Direct Operands (d indicates a displacement specified in the instruction)
Optional DR modification by any one of these items
Data segment (or stack segment)
DR
(b) Access via DR
Optional modification by B
(c) Indirect Descriptor Access

Figure 5.2 Operand types for arithmetic and logical operations

of the locations indicated by forms 2 to 5 above and the descriptor in this case may be optionally first modified by the contents of B. These cases are illustrated in figures 5.2b and c.

Arithmetic and logical functions operate between ACC and another operand and are provided uniformly for all those data types and for all sizes of individual data types for which the operation has meaning. Arithmetic orders are provided for: add, subtract, multiply, divide, reverse subtract and divide, remainder divide, double-length multiply and divide, compare and arithmetic shift. There are also the logical functions: and, or, not-equivalent, rotate, shift and shift-while-zero, as well as two-way conversions between fixed and floating and fixed and decimal data.

In addition to these ACC operations, some arithmetic and logical functions can be performed in B and DR, other registers can be loaded and stored in various ways and there is the usual complement of comparison and jump instructions.

Store-to-store instructions operate either between a byte and a byte string referenced by a descriptor or between two such strings. Facilities are available for scanning, comparison, moving, table look-up, insertion, logical operations, packing, unpacking and editing.

It can be seen that this general function code satisfies the cleanliness requirements specified by section 2.3. However, there are in addition special-purpose instructions, which, while not of general use in high level language translations, are included under the criterion that they have a significant effect on over-all system performance. An example is the validate instruction, which can check whether a descriptor provided to a called procedure is valid at the access level of the caller (see section 5.4).

5.3 VIRTUAL STORAGE AND INTERRUPTS

The 2900 virtual byte address is 32 bits long. It is split into 14 bits for a segment number and 18 bits for the byte displacement within the segment. Thus any virtual machine has a maximum of 2^{14} segments, each of a maximum of 2^{18} bytes. To recap on section 4.3, the mapping information between virtual and real storage is contained in a segment table. Hardware access to this is facilitated by a *segment table base register*, STBR, which contains the real store address of the first item of the segment table of the currently active process, together with the length of the table. Each segment table entry is eight bytes and contains the real address of the segment, the length of the segment, its protection status and whether it is present in main store or not.

The practical problems of real-to-virtual mapping are, however, too complex to be handled by the simple mechanisms we have considered so far. For example, some of the problems that arise are as follows

- segment tables for each segment in each virtual machine would rapidly become too large to keep in main store;
- sharing segments by holding copies of the real address in each segment table

gives a great overhead on moving the segment, since all references to it in all virtual machines must be found and altered;

- variable length segments can, as we have seen, lead to store fragmentation, which is worsened by the overheads of moving;
- access via an in-core segment table means that every variable reference requires two store accesses, one for the segment-table entry and one for the variable itself; this is an unacceptably large overhead.

In any practical virtual store implementation, solutions must be found to these problems. Those that are described in this section are a mixture of range standards and implementation choices.

First, we already know that a large number of segments belonging to what on a conventional machine would be the operating system will be included in all virtual machines. These are called *public segments*. We arbitrarily assign these the same segment numbers in the range 8129 to 16 383 in each virtual store (that is, segment numbers with a 1 in the top bit). We can thus have a separate segment table for these items, which is common to all processes. This reduces considerably the overall size of segment tables.

Second, we can separate all segments that are not public but are shared by two or more processes and put their addresses into a *global segment* table. The corresponding entries in the segment tables of the processes that share such segments now contain a bit indicating that they are shared and, instead of a real address, a reference to the global table entry. This is shown schematically in figure 5.3. (This indirection mechanism could be taken to any level if desired.) There is thus only one copy of each real address to be altered if a segment is moved. In a multi-access environment, where several users are accessing the same compiler, for example, local segments would be confined to data areas, the compiler code being global.

There is no complete known solution to fragmentation but we can (at least on medium-to-large machines with many simultaneous processes) divide the segment further into fixed-length *pages*, which are the unit for transfer between the store hierarchies. There will thus never be external fragmentation since all transfer units are pages and all holes are of page size. However there will now be internal fragmentation–storage wasted at the end of each segment since the segment will not in general be an integral number of pages. This wasted fraction must statistically be $p/2s$ per segment where p is the page size and s the average segment size (that is, on average half a page is wasted per segment). To minimise this we can either minimise p or maximise s. If p is small, however, we need very large *page tables* (the equivalent of segment tables but for pages). So we must maximise s, which is a reason for grouping program areas of similar types rather than mapping them one-for-one on to segments. We chose 1024 bytes as a reasonable page size and a virtual address thus takes the form

Segment	Page	Byte address
14 bits	8 bits	10 bits

As a further compromise we can allow unpaged segments as well as paged in the same process, the relevant segment table entry being marked accordingly. This

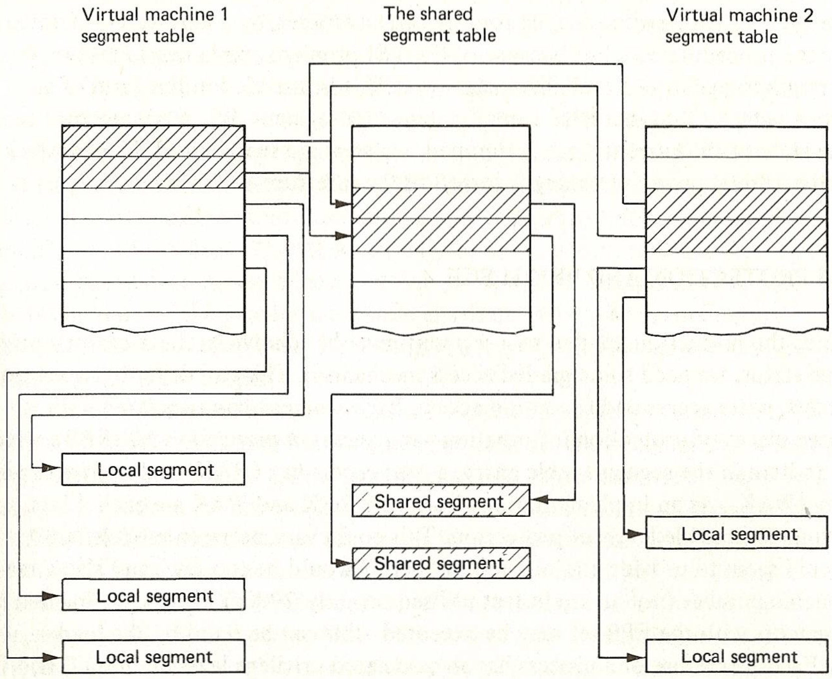


Figure 5.3 Segment sharing

facility is mainly used for very small segments (especially smaller than one page) particularly in the system software, or for very large segments that are permanently in store.

The extra indirections introduced by the global segment table and paging have worsened the access overhead. To overcome this we introduce *current page registers* (CPRs) and *current segment registers* (CSRs), which hold the most recently or most frequently used table entries and can be searched in parallel by hardware to avoid any unnecessary store accesses. The number of such registers and the complexity of the algorithm by which they are updated are a matter for cost–efficiency trade-off for any particular machine.

Interrupts are quite complex in the chosen architecture. They are basically of two types. The first are concerned with autonomous-unit synchronisation: external stimuli, multi-processor messages and peripheral interrupts. The subtypes differ considerably with information being passed in buffers, lines or main store. The second type are more concerned with the actual CPU design: system errors, timer interrupts, virtual store interrupts, system calls, extracodes (function codes not in fact supported by hardware but requiring software subroutines), program errors or exception conditions. This collection is messy in that greatly different techniques are needed to deal with each. A mechanism is thus required to analyse the type of interrupt, localise its peculiarities, determine its priority and call for the necessary services. This *interrupt decoder* is in public segments (which implies co-operation between virtual machines so that one process can carry out an interrupt service for another). The interrupt-decode routine can be entered by a hardware-generated, forced procedure call, but because of the VSI problem, explained in section 4.5, it requires guaranteed available main store. This is provided in the form of a second stack that is actually a small public-data segment. When an interrupt occurs the state of the current stack is dumped, registers are switched to the new stack and a normal procedure entry is forced to the interrupt decoder.

5.4 PROTECTION AND PRIVILEGE

Since the protection applied to a segment must be relative to the accessor's privilege status, we need some graded access mechanism. This can differ between read access, write access and execution access. Each segment has associated with it three pieces of protection information—an *execution permission bit* (EPB), which is an item in the segment table entry, a *read access key* (RAK) and a *write access key* (WAK). As an implementation choice the RAK and WAK are each 4 bits, giving 16 possible levels of protection. This could vary between models but it would seem to provide a minimum number (7 would be too few) and also a reasonable number (not all are in fact utilised on early 2900s). Only code located in segments with the EPB set may be executed—this can be fixed by the loader.

Each procedure of a process has an associated privilege level between 0 and 15. The software of a process can be distributed through these levels in any convenient way but on existing implementations a convention is adopted for subsetting them. Three levels are devoted to the kernel, which, as we have seen, handles real resources. The remainder of the system software is structured between seven levels, and six levels are left for user applications. These last can be used, for example, to allow procedures under test to be tried out in an operational, on-line user system while still protecting vital system code and data.

When the process is active a hardware register, the *access control register* (ACR), holds the privilege level of the currently executing procedure. A particular segment can be accessed for reading if, and only if, the ACR value is less than or equal to its RAK. The segment may be accessed for writing if, and only if, the current ACR value is less than or equal to its WAK. Thus a low number indicates high privilege and high protection status. (This can be a pitfall for the unwary; readers are requested to consider the context of 'high' and 'low' in what follows very carefully.) Whenever a store access is attempted a hardware check is made of the relevant segment-access key against the current setting of the ACR. Since the check is not for equality it means that procedures are increasingly trusted as their privilege increases. A particular segment may therefore be inaccessible to one privilege level, accessible only for reading to a higher level and accessible for reading and writing to a higher level still. This facility is heavily utilised within the supervisory software, so that, for example, peripheral handling routines can access data tables connected with their peripherals and thus have direct control of real resources, but are nevertheless forbidden access to data concerned with store allocation. This level structuring also assists in localising program faults and preventing their propagation throughout the system, thereby permitting more rapid diagnosis and higher chances of automatic recovery.

The ACR register forms a part of a larger hidden register called the *program status register*, PSR. Also in the PSR is a *privilege bit* PRIV. Only if PRIV is set to 1 can the ACR be changed by instruction and this facility is reserved for the most trusted part of the kernel. The ACR value is dumped and restored on interrupts and system calls so that these provide the mechanisms for increasing the current level of privilege. The ACR can only be set to a less privileged level by exiting from such routines. Access keys are set by the kernel and not altered. The remaining items of the protection system are dealt with by such supervisory subsystems as the loader. Thus, for example, a segment can be loaded with different privilege at different points.

We are now in a position to examine more detailed implementation matters that are essential to a fool-proof system. First consider the problem that although a non-privileged procedure cannot access restricted data it can, via a system-call descriptor, invoke a procedure that can. This is an essential part of the process operation but is obviously open to misuse. When a system-call descriptor is encountered, a system-call interrupt occurs. The system-call descriptor contains a reference to an entry in a *system-call table*, which defines the called, or target, procedure. In this entry is stored the ACR of the target and a number, K, which indicates the maximum ACR value of the class of procedures that are allowed to call it. Direct call of trusted routines can thus be forbidden to less trusted code. The process-interrupt manager will allow a requesting procedure to change a K value to any value higher than its own ACR level. This allows a supervisor procedure to isolate another kernel facility from less privileged code.

Figure 5.4 illustrates how the protection mechanisms work. We consider a simplified process with three code segments. In increasing order of privilege, these segments are: A, which has access level 10; B, access level 7; and C, access level 4. The ACR values shown are those that would be found when the associated segment was executed. There will also be within the process some kernel code, which is not illustrated but has access level 0. The process has data segments D, E, F with differing Read Access Keys and Write Access Keys. Two further segments used by the kernel, the System Call Table, G, and the segment table, H, have RAK and WAK equal to 0, as do all the code segments. All this information is contained in the segment table H, together with the execute-permission bits.

We can see immediately that only the code segments A, B, C may be executed. Also, since both RAK and WAK are zero for code segments, only the kernel may read or write to them. This restriction also applies to the tables G and H. Data segment D has RAK = WAK = 10, which means that A, B, C or the kernel can access D for reading or writing. F has RAK = WAK = 4 and can therefore be read or written to only by C or the kernel. It is not accessible directly to A or B. Segment E on the other hand has different RAK and WAK settings. The RAC value of 7 forbids reading by segment A but allows B or higher levers to read it. However, the WAK value is 1 implying that this data is read-only except at the two highest levels of privilege within the kernel.

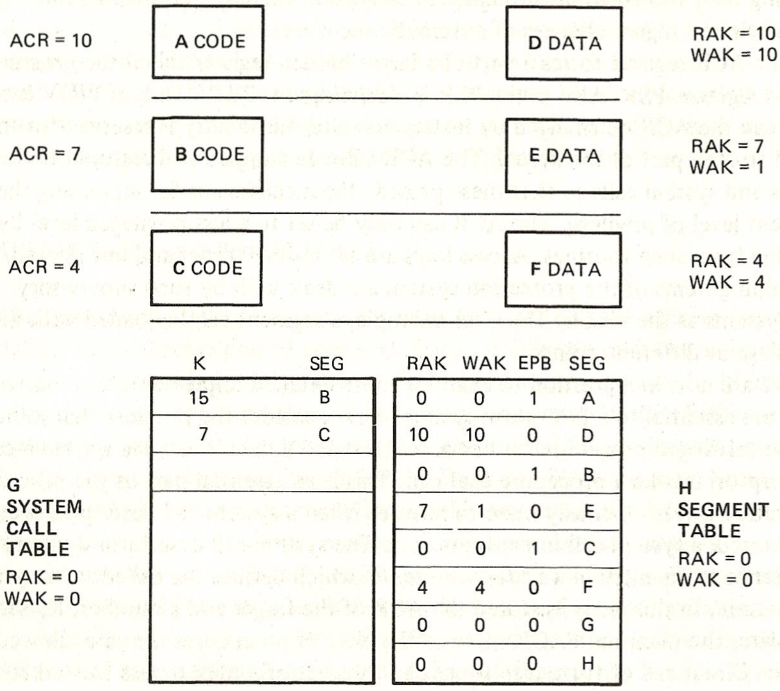


Figure 5.4 Protection mechanisms

Note that special provision needs to be made to allow the access of literal operands and constant values stored within the code (see section 5.2). Such access is only permitted to the code segment in which they occur.

The system call table, G, contains the procedure-call permission list for the virtual machine. The two entries illustrated show that, while B can be called from any ACR level, which includes an 'inward' call from A or an 'outward' call from C, C itself can only be called by procedures with ACR less than or equal to 7. This will permit a call by B but not by A. The only way in which A can obtain a service from C is therefore indirectly, via a system call to B.

The choice between constructing a system-call descriptor or a normal procedure descriptor as the operand of a call instruction is made by the loading subsystem, which is aware of the privilege level assigned to the segment being loaded and of other segments in the system. A procedure descriptor will contain a normal virtual address. A system descriptor on the other hand contains an index to the *system call table*, which will be used by the system-call interrupt routine to determine whether the call is legal. Normal procedure descriptors are used for calls to routines at the same ACR level and are also constructed dynamically as part of the link information for a return of this type. Such a link descriptor incorporates program-status information—for example, the ACR value and whether arithmetic overflow has occurred as well as the return address. In the case of an *inward system call*, that is, a call for a procedure of greater privilege, a system-call descriptor will be used to invoke the required procedure and a normal procedure descriptor can be used for the link since no special action will be needed.

The much less frequent case of an *outward system call* is more complex. In this case we have a less privileged routine being invoked and in general such a routine cannot be trusted not to interfere with vital information on the stack. The system-call interrupt routine will in such circumstances prepare and activate a new stack segment and protect the old one before invoking the less privileged procedure. In this case the link constructed will be another system-call descriptor so that the old stack, program status and system status can be restored on return. In all cases—normal, inward or outward,—the pre-call and post-return sequences are identical. It is also worth noting that on the 2980 the entire inward-call sequence is implemented in hardware.

The software-system implementation adopts an interesting method for dealing with the storage of inter-segment references so as to preserve the full generality of sharing. While not strictly a 2900 architectural matter the principle is worth some investigation. Consider the case of a procedure that is to be shared between processes. This procedure calls a private procedure, which will differ for each process, for example, a user-specified error routine. Where is the descriptor (procedure or system call) of the private routine to be stored? If it is stored in the code segment and accessed by operand type (1) or (2) of section 5.2, the code can only be shared if the private procedure has the same virtual address in all processes. This is possible to organise but messy. Alternatively if the descriptor is stored in any static data area—either in a data segment or in a fixed position relative to the outermost global namespace on the stack, say—the same virtual address or displacement within the global space will have to be used for the descriptor itself within each sharing process, which is essentially the same problem. The ideal place is obviously in the local namespace of the shared procedure but this only exists when the shared procedure is invoked.

The following method has been adopted to overcome this difficulty. All descriptors of external items of this type associated with a procedure are collected into a *program linkage table*, PLT. The relative position of each descriptor within this table is known to the procedure and the PLT is stored off-stack in a static location. A descriptor that points to the PLT is by convention passed to every procedure as its first parameter. By using this descriptor, modified by the relative position of the required descriptor within the PLT, the procedure can access any of them without its code containing any fixed addresses and the procedure, or any of the items that it references, can be assigned any virtual address in any process without reducing sharing. To reduce the overheads of indirection the *linkage table base* register, LTB, is used, by convention, to hold the address of the PLT. It is set from the first parameter of the procedure. Data segments can obviously be dealt with in the same way as code segments.

The reader may be asking how a calling procedure knows the PLT address of a procedure that it calls. Several possibilities are available depending on implementation considerations such as space, time and the availability of registers like LTB. An obvious way is to extend the PLT so that, for each procedure entry, not only the code descriptor but the PLT descriptor for that code is stored. This would solve the problem without affecting sharing but at a PLT-space cost. Alternatively, this space can be traded for an extra indirection by arranging that the PLT holds *only* the descriptors for the PLTs of the accessed routines and that each PLT conventionally has as its first entry the code descriptor of the associated code itself. The situation is then as in figure 5.5.

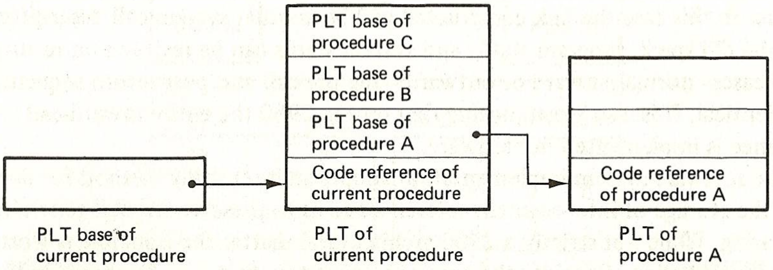


Figure 5.5 Possible PLT operation

To access procedure A from the current procedure, A's PLT-base address can be accessed by a modified use of the current PLT-base descriptor. This is placed on the stack as A's first parameter. An indirect access via this descriptor can then be used as an operand for a call instruction to invoke A. It is left as an exercise for the reader to see how system calls can be handled in this situation. It is, after all, an implementation rather than an architectural matter but it illustrates the power and flexibility of some basic architectural features.

The protection system allows subsystems to provide services directly to less trusted procedures while still protecting data. However, since virtual addresses are merely binary patterns there is nothing to prevent any routine constructing any address. (This is in contrast to the Basic Language Machine, where addresses are specifically tagged and can only be constructed by highly trusted routines.) Attempts to access forbidden data using an illegal address constructed in this way will fail, because of ACR protection. Nevertheless the criminal or negligent procedure could pass such an address as a parameter to a more privileged routine at a system call, and thus gain illegal access indirectly. On a system call, therefore, it is necessary to check that any descriptor passed as a parameter was valid at the ACR level of the caller. This is quite easily done but since we require system calls to have low overhead to combat the conventional OS service problems, a hardware validate instruction is provided to speed up this operation.

Finally the protection mechanism described only effectively protects items in store. Other real resources such as peripherals also need protection; random application programs cannot be allowed to drive peripherals directly by error or design.

This, however, is much simpler to control, since direct peripheral access can be restricted to specific routines. The protection is in fact managed by an extension of the use of the *program status register*, PSR, which allows specific routines to access directly machine hardware registers. These registers are treated as though they represented a highly protected piece of store, the *image store*, which can be accessed only by procedures with the correct PSR settings. Special operand forms that indicate an image-store access are allowed, but these can be used by the privileged routines together with normal manipulative functions.

6 *Initial Implementations*

This chapter describes some of the important implementation features of the first 2900 hardware and software systems realised by ICL. There are essentially three processors, the 2960, 2970 and 2980 although hybrid systems with the processing power of one of these and connectivity of another are also available. There are also two supervisory systems or Virtual Machine Environments: VME/B, which is available on all machines, and VME/K, which was designed for 2960. Also of interest, because of the initial range objectives, are the ways in which data management and peripherals have been treated. Each of these subjects is dealt with superficially in this chapter. Of all the chapters this is the least complete description both because more detailed information can be easily discovered in ICL literature and because the subject matter is essentially of transitory interest, since if the 2900 is to meet all its objectives as a range of machines one can expect additions and improvements to the range for a considerable time to come.

6.1 PROCESSORS

All the basic processor models have adopted a structured approach to the provision of the processing system. Each processor consists of a number of basic modules. These are illustrated in figure 6.1. First there are one or more order code processors (OCPs). These are the units concerned with basic arithmetic and logical operations. Main store is provided in semiconductor blocks and is accessed via a Store Multiple Access Control (SMAC). Each block is independent and can be accessed simultaneously. Peripheral controllers access the main store via a Store Access Control unit (SAC) which, as far as the store is concerned, is functionally equivalent to an OCP. The OCP is thus not concerned with handling peripheral transfers, which after initiation are dealt with autonomously by the SAC. Figure 6.1 shows the cross-connection of both OCP and SAC to both the SMACs. This basic structure leads naturally to multiprocessor systems, which can provide improved throughput with sufficient redundancy to allow reconfiguration in case of component failure. A typical multiprocessor configuration is shown in figure 6.2.

All models employ ICL 1000 and Schottky TTL with ICL's matched interconnection technology. The larger machines employ platters of up to 20 layers while the 2960 uses up to 17, each supporting macro circuit boards. Each model employs pipelining (instruction overlap) and slaving techniques (see section 5.2) to improve throughput, the degree of use varying according to the individual cost–performance objectives of the processor. Considerable attention has been

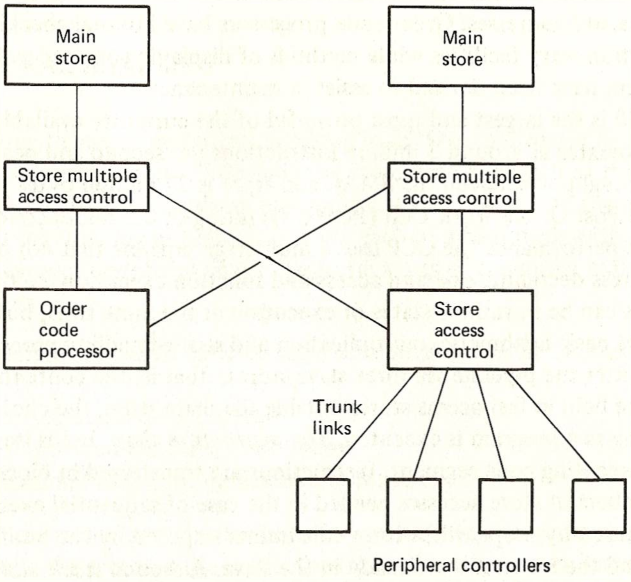


Figure 6.1 Processor structure

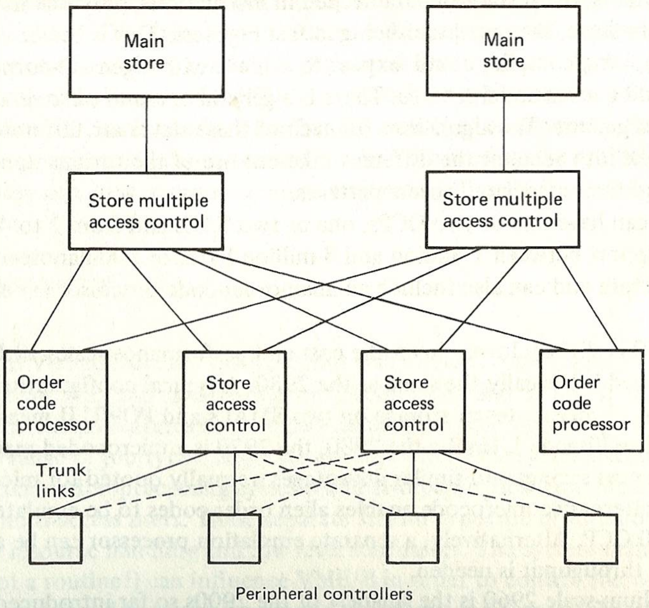


Figure 6.2 Multiprocessor configuration

paid to reliability, both in the selection of the technology, the reconfiguration ability mentioned above, and by selective hardware self-correction techniques. The SMAC performs Hamming-type checks on the main store while the SACs parity check all data received from the SMAC or the trunk links including control data and addresses. Order code processors have internal checking systems and instruction retry facilities while methods of displaying and recording the status of the system have been devised to assist in maintenance.

The 2980 is the largest and most powerful of the currently available systems. The OCP operates at around 3 million instructions per second and peak instantaneous throughput between the SMAC and store is 27 million bytes per second. This gives a Post Office Work Unit (POWU II) rating of 0.3 milliseconds. To achieve this performance the OCP uses a multistage pipeline that not only has phased address decoding, operand access and function execution, so that several instructions can be in various states of execution at the same time, but also has independent basic arithmetic, multiplication and string-handling processing units. Items can enter the pipeline via three slave stores; that is, the contents of various locations are held in fast-access stores outside the main store, the choice of locations varying as a program is executed. The *instruction slave* holds items from the currently executing code segment. Instructions are transferred in blocks, reducing the numbers of store accesses needed in the case of sequential execution, while branches caused by loops will, within constraints imposed by the number of slave registers, find the instructions already in the slave. A second *stack slave* holds accessed items from the stack, the most commonly used segment of any process. This is a particularly cost-effective slave store, a hit rate of 90 per cent being common; that is, of all stacked data needed in instructions, only one in ten is fetched from store, the remainder being in fast registers. This is better than even a highly optimising compiler could expect to achieve with a general-purpose register structure and conventional storage. There is a general *operand slave* on items from other data segments. The algorithms for each of these slaves are different since they can take into account the different inherent use of the various items in memory and their associated access patterns.

A 2980 can have one or two OCPs, one or two SACs and from 2 to 4 SMACs. This can support between 1 million and 8 million bytes of 500-nanosecond semiconductor store and can also include an alien-order-code processor for emulation purposes.

The 2970 is slightly lower down the cost and performance scales, although the store speed is basically the same as the 2980. A typical configuration might have around $1\frac{1}{2}$ megabytes of storage on two SMACs and POWU II measurement of around 1 millisecond. Unlike the 2980, the 2970 is a microcoded machine. Apart from cost savings and similar advantages normally quoted for microcoding, loading an alternative microcode enables alien order codes to be emulated directly on the 2970 OCP. Alternatively, a separate emulation processor can be attached if the extra throughput is needed.

The medium-scale 2960 is the smallest of the 2900s so far introduced. It uses 850-nanosecond store, two-stage pipelining and less ambitious slaving techniques to achieve a POWU II measurement of the order of 2 milliseconds. 2960, like 2970, is a microprogrammed machine capable of directly emulating alien order codes.

6.2 PERIPHERAL HANDLING

The underlying theme of peripheral handling in 2900 is to disperse intelligence throughout the system, thus continuing the structured approach of the processors out into the system. The *peripheral controllers*, which connect to the SAC via a standard trunk link, are autonomous units, each of which deals with a number of a particular type of peripheral device. Separate types of controller exist for the following units

- general peripherals: serial devices, input and output peripherals and magnetic tape
- disc files: exchangeable disc stores
- sectored files: fixed-head discs and drums
- communication links: communications systems

The disc file controller contains facilities for rotational position sensing, queueing and retry while the sectored file controller takes advantage of the known sectored structure of the file to provide the sort of performance needed for efficient one-level store implementation.

Each controller consists of a central logic unit plus one or more peripheral interface modules. This eases connection of existing peripheral equipment and introduces some flexibility for future peripheral developments. In the case of the communications link controller (CLC) the interfaces are known as Network Interface Modules, each of which can handle a number of channels. The CLC is in fact a microprogrammed processor and can be enhanced by the addition of storage and provision of a new microprogram to become a full-scale front-end processor.

6.3 SUPERVISORS

The VME/B Virtual Machine Environment was the first supervisory system to be announced by ICL and is available on 2960, 2970 and 2980. It exploits the architectural design features to provide a general-purpose operating environment for a wide range of job types. Separate options within the system handle and schedule transaction-processing systems and also batch jobs from local or remote sites and multi-access users. These separate stream types are in turn co-ordinated by central resource handling and low-level schedulers. The system manager (a human, not a routine!) can influence VME/B in order to control the way in which resources are utilised between these various streams or between jobs within a stream.

The over-all system is controlled by a *system control language*. This is in fact a purpose-built high-level block-structured language with variables, computational facilities, conditional statements, etc. The block structure allows the phasing of a total job and the acquisition and release of resources to be stated in a straightforward manner. However, macro facilities and default options allow simple or frequently run jobs to be initiated using a very simple interface.

VME/B devotes considerable attention to resilience by fully exploiting the ACR system within the supervisor and by providing error-management, correction and diagnosis routines at various levels within the system. This, together with system calls and parameter validation, is aimed at containing hardware or software faults wherever possible and preventing error propagation.

VME/K is offered by ICL on 2960s and has a simpler structure more suitable for the applications area at which it is aimed. Like VME/B it is capable of dealing with various work-load types but it handles them in a different way. Separate *function processors* are devoted to batch, MAC and TP, which exist in the virtual store of each process of the appropriate type. Additional function processors handle such matters as operator communication, emulation, and local and remote spooling. Each function processor is responsible for resource control within its own application area and all are co-ordinated by the central controlling routines, known collectively as the Executive. Basic mechanisms are provided for manipulation of programs and data within system files and special streamlined access is allowed for TP system users. Job control is via a more conventional job control language (JCL).

Both operating systems support the same set of compilers and utilities including specific utilities concerned with transfer of programs and data from other systems.

6.4 DATA MANAGEMENT

The virtual machine concept particularly benefits the implementation of data-management systems. On conventional systems, the data-management system either sits between the programs and the operating system, introducing access overheads, or exists in each program that needs it, introducing space overheads; on the 2900, by contrast, data-management routines can sit in the same virtual store as the 'program' and the operating system, but are shareable.

On VME/B separate routines are provided for different record-access mechanisms —serial, random, indexed sequential, etc.—and those corresponding to the files needed by the user are linked into his process at load time. On top of these basic access mechanisms, it is possible to introduce other routines that will map the user's view of a file on to the actual format it takes and thus provide virtual files. The way in which the user interacts with the system is the same at any level of virtuality.

With VME/K the basic accessing methods are provided within Executive, which is also responsible for fundamental file creation. Both VME/B and VME/K have a form of *data description language*, which allows the format and content of a file to be described and stored for future use. In this way a unified interface can be provided to all data, and degrees of data and program independence can be introduced, since the actual file definition can be stated outside the program proper.

The way in which data source is concealed from the user in both VMEs is particularly useful in the case of communications handling. Applications programs are shielded from the characteristics of terminals and communications sytems, and data bases can interface easily to each other.

A range of data-management facilities is provided from a set of utilities through to an Integrated Data Base Management System, and in accordance with the fundamental design principles the user is allowed to choose his level of complexity in this area and make his own balance between system power and overhead costs.

6.5 PORTABILITY

It has been noted in the previous sections of this chapter that all aspects of hardware and software implementation have been constrained by the need to allow easy transfer of user jobs, data and equipment from existing machines to 2900. Thus, processor configurations support alien order-code processors and/or microcode emulation; controllers and peripheral interface modules allow the incorporation of alien devices; while the supervisory, utility and data management software provide support or translation facilities for foreign programs and data. It is interesting to note that the microcode emulation facilities have been so successfully implemented as to allow ICL to take a *post hoc* marketing decision to offer large 1900 customers the option of using a 2900 merely as a super 1900 (the so-called Direct Machine Environment or DME)!

7 *Conclusions*

It is too early as yet to say what the 2900's place in computer history will be. Systems can be successful in their own terms while never selling in great numbers. Multics, the Basic Language Machine and Atlas all added considerably to computer-science theory and techniques without ever being runaway commercial successes. On the other hand ICL must judge success in terms of the numbers of systems sold and profitability. While initial signs look good, it will be some time before such measurements can be accurately taken. This form of success is also subject to marketplace intangibles, over which the system architect has no control whatsoever.

At this stage then, the only measure of success of the Synthetic Option Team and their successors is obtained by comparison of the eventual design of the 2900 systems with the original objectives of chapter 2. This must be a subjective judgement on the part of each reader but, in order to help (and possible bias) him, figure 7.1 provides a mapping of the objectives on to the features of the system. The list at the left-hand side represents the major aims outlined in chapter 2. They are grouped rather loosely under major headings but, for example, many of the 'high level interface' features could also appear under 'user application effectiveness'. Each column represents one of the important features of the architecture or implementation described in chapters 3 to 6, again grouped under major headings. Wherever an architectural feature has a significant impact on a particular aim, the intersection of the corresponding rows and columns is marked. Otherwise it is left blank. Thus, for example, by reading down the column marked 'Interrupt handling', we can see that this group of architectural features has an impact on system throughput, communications, the match between supervisor and hardware, transaction processing, multi-access computing and resilience. Conversely, reading across the row marked 'Good procedure handling' we can see that this is provided by segmentation and the stack.

Any attempt to chart such a complex mapping is bound to be subjective, and so the reader is invited to consider what entries he would consider important. The list of aims also omits a fundamental objective set out in chapter 2, that the architectural design of the Series should be coherent and first-class technically. On this matter the author declares an interest, and leaves the consideration as an exercise for the reader.

		ARCHITECTURE															
		Virtual machine				Process			Program structure		Primitive architecture		Implementation				
AIMS		Virtual processing	Virtual peripherals	Virtual store	Segmentation	Process structure	Interrupt handling	Protection system	Stack	Descriptors	Instruction formats	Registers	Processors	Peripherals	Supervisors, etc.	Data management	Dispersed intelligence
Internal cost reduction	Hardware modularity										●	●	●	●			●
	Software modularity		●		●				●						●	●	
	Technology exploitation								●	●	●	●	●	●			●
User application effectiveness	Shared code	●	●	●	●	●		●	●	●	●	●			●	●	
	System throughput	●				●	●						●	●	●		●
	Processor speeds										●	●	●				●
	Data speeds												●	●			●
	Communications		●	●	●		●							●		●	●
High level interfaces	Data management		●							●				●		●	
	Data independence				●				●	●	●					●	
	Storage type independence		●	●	●											●	
	Hardware/supervisor match	●				●	●		●		●		●		●		
	Clean order code								●	●	●	●	●				
	Good procedure handling				●				●								
	Structure handling				●					●	●	●					
	Expression evaluation								●		●	●					
	Object code support		●	●		●									●	●	
Versatility	Modes of use					●		●							●		
	Batch	●	●	●	●			●							●	●	
	TP	●	●	●	●		●	●						●	●	●	●
	MAC	●		●	●	●	●	●							●	●	
Transfer	Programs	●											●		●		
	Data		●											●	●	●	●
	Equipment												●	●	●	●	●
Dependability	Reliability					●		●					●	●	●		●
	Resilience					●	●	●					●	●	●		●
	Security	●	●	●	●	●		●							●		

Figure 7.1 Impact matrix

Appendix A Order Code Structure

A.1 INSTRUCTION FORMATS

Instructions may be 16 or 32 bits long. There are three formats in all, of which the first 7 bits specify the function, the remaining 9 or 24 bits the operand. The format is determined by function code and the length by the operand code. There are 104 primary format instructions, 16 secondary format or store-to-store instructions and 8 tertiary or jump instructions.

A.2 PRIMARY OPERAND DECODE

The allowable operands consist of literals within the instruction, direct operands accessed via a pointer register, or indirect operands accessed via a descriptor. The conventions used below to describe the operand forms are illustrated by the following

n	a 7-bit literal
N	an 18-bit literal
LNB	the address contained in the Local Name Base Register
XNB	the address contained in the Extra Name Base Register
LTB	the address contained in the Linkage Table Base Register
PC	the address of the current instruction (contained in the Program Counter)
B	the modifier in the Index Register
DR	the descriptor in the Descriptor Register
TOS	the item at the top of the stack
IS	the Image Store (see section 5.4)
(LNB+N)	the data item in LNB+N (that is, the data item displaced N words from the item referenced by the Local Name Base Register)

((LNB+N))	the data item referenced by the descriptor (LNB+N)
((LNB+N)+B)	the data item referenced by the descriptor (LNB+N) after the descriptor has been modified by B
(DR+(LNB+N))	the data item referenced by the descriptor DR after the descriptor has been modified by (LNB+N)

A.2.1 16-bit Form

The 9 bits of the operand are decoded as follows

	K	n
bits	2	7

The meaning depends on the value of K as follows

K	*operand*
0	the 7-bit signed literal n
1	(LNB+n), n unsigned
2	((LNB+n)), n unsigned
3	n is decoded further as follows

	K1	K2	R
bits	2	3	2

R	is reserved and not used
K1	may take the values 0–3
K2	may take only the values 6 or 7

The meaning is as follows

	DIRECT	INDIRECT		
		DESCRIPTOR IN DR, MODIFIED	DESCRIPTOR IN STORE	DESCRIPTOR IN STORE, MODIFIED
K1 \ K2	0	1	2	3
6	TOS	(DR+TOS)	(TOS)	(TOS+B)
7	B	Unassigned	(DR)	(DR+B)

A.2.2 32-bit Form

The 25 bits of the operand are decoded as follows

	K	K1	K2	N
bits	2	2	3	18

K	is always 3
K1	may take values 0 to 3
K2	may take values 0 or 2 to 5 (1 is unassigned)

The meaning is as follows

	DIRECT	INDIRECT		
		DESCRIPTOR IN DR MODIFIED	DESCRIPTOR IN STORE	DESCRIPTOR IN STORE MODIFIED
K1 / K2	0	1	2	3
0	N signed	(DR+N)	Note 1	Note 2
2	(LNB+N)	(DR+(LNB+N))	((LNB+N))	((LNB+N)+B)
3	(XNB+N)	(DR+(XNB+N))	((XNB+N))	((XNB+N)+B)
4	(PC+N)	(DR+(PC+N))	((PC+N))	((PC+N)+B)
5	(LTB+N)	(DR+(LNB+N))	((LTB+N))	((LTB+N)+B)

Note 1. Privileged direct operand–IS location N.
Note 2. Privileged direct operand–IS location B.

A.3 SECONDARY OPERAND DECODE

This format is used for store-to-store operations. The operand field of the 15-bit form and the first 9 bits of the 32-bit form are equivalent, and the 32-bit form has two extra 8-bit fields as follows

16-bit form	H	Q	n		
32-bit form	H	Q	n	M	L
bits	1	1	7	8	8

The difference between the two forms is determined by the value of Q

Q = 0	16-bit form
Q = 1	32-bit form

H determines the length of the string in the operation

if H = 0, length = n + 1

if H = 1, length = length from descriptor of destination string

M is used as a mask byte, L as a literal or filler byte, which can replace the source string.

A.4 TERTIARY OPERAND DECODE

This format is used for conditional jump instructions

16-bit form	M	K3	R
bits	4	3	2

32-bit form	M	K3	N
bits	4	3	18

M is a 4-bit mask field.
R is unused and should be set to zero.
K3 provides the following operand types

K3	*operand*	
0	N literal	32-bit form
1	(DR+N)	
2	(LNB+N)	
3	(XNB+N)	
4	(PC+N)	
5	(LTB+N)	
6	(DR)	16-bit form
7	(DR+B)	

A.5 INSTRUCTION LIST

The following list merely names the instruction except in those cases where the function is not immediately obvious from its name, when explanatory notes are provided.

A.5.1 Control Instructions

		Notes
Load LNB	Store LNB	
Load XNB	Store XNB	
Load LTB	Store LTB	
Adjust SF	Store SF	
Raise LNB		
Increment and Test	Test and Decrement	1

Note 1. Semaphore instructions.

A.5.2 Jump Instructions

		Notes
Call	Exit	
Jump	Jump and Link	
Decrement B and jump if non-zero	Jump on condition code	
Jump on arithmetic condition true	Jump on arithmetic condition false	
Escape exit	Idle	
Out		2

Note 2. 'Out' forces an interrupt.

A.5.3 B Instructions

		Notes
Load B	Stack and Load B	
Store B	Add to B	
Subtract from B	Multiply B	
Compare B	Compare and Increment B	
Dope-vector multiply		3

Note 3. 'Dope-vector multiply' assists with the calculation of array elements for arrays stored in FORTRAN-like form.

A.5.4 DR Instructions

		Notes
Load DR	Stack and Load DR	
Store DR	Modify DR	
Load bound field	Load address field	
Load type and bound fields	Increment address field	
Validate address		4
Load relative		5

Note 4. See section 5.4.
Note 5. See section 5.1.

A.5.5 ACC Instructions

		Notes
Set ACS 32 and load	Set ACS 64 and load	6
Set ACS 128 and load	Stack, Set ACS 32 and load	
Stack, Set ACS 64 and load	Stack, set ACS 128 and load	
Stack and load	Load	
Store	Load upper half	
Store upper half	Copy DR	
Modify PSR	Copy PSR	7
Read real-time clock		

Note 6. Accumulator-size register, see section 5.2.
Note 7. Program-status register, see section 5.4.

A.5.6 Computational Instructions

Computational operations in the accumulator are affected by the previous setting of ACS. Many are common to all arithmetic-data types as shown by the following table

	Floating	Fixed	Logical	Decimal
Add	x	x	x	x
Subtract	x	x	x	x
Reverse subtract	x	x	x	x
Compare	x	x	x	x
Shift (scale)	x	x	x	x
Multiply	x	x	–	x
Divide	x	x	–	x
Reverse divide	x	x	–	x
Remainder divide	–	x	–	x
Divide double	x	–	–	–
Multiply double	x	x	–	x

x Provided
– Not provided

The following instructions peculiar to one arithmetic type are also available:

Additional floating-point instructions
- Fix

Additional fixed-point instructions
- Convert to decimal
- Float

Additional logical instructions
- And
- Or
- Not equivalent
- Rotate
- Shift 32 bits
- Shift while zero

Additional decimal instructions
- Convert to binary

A.5.7 Store-to-store Instructions

The following instructions operate in general either between two designated strings or between one string and a literal byte. A second literal byte may also be in some cases specified as a mask or a filler.

Scan while equal	Table translate
Scan while unequal	Pack
Compare strings	Suppress and unpack
Move	Conditional insert
Check overlap	And Strings
Move literal	Or strings
Table check	Not-equivalent strings

Appendix B Data Formats

B.1 FLOATING-POINT FORMAT

The formats for 32-bit, 64-bit and 128-bit floating-point numbers are compatible, in the sense that the first word of a 64-bit number is a legal 32-bit number and the first two words of a 128-bit number form a legal 64-bit number.

The formats are as follows

	S	Biased exponent	Fraction	Continuation of Fraction
Bits	1	7	24	32

	Ignored	Continuation of Fraction
Bits	8	56

S is a sign-and-modulus sign bit, 0 = positive, 1 = negative. Biased exponent is the true hexadecimal exponent + 64. Bits 8 onwards contain an unsigned fraction. For 128-bit form, bit 72 is assumed effectively adjacent to bit 63.

B.2 FIXED-POINT FORMAT

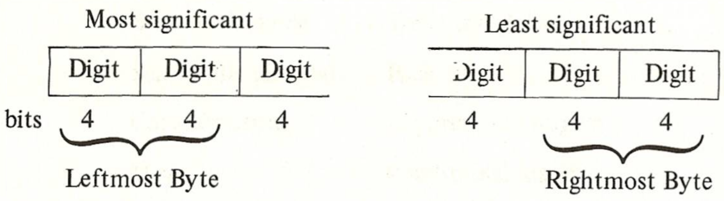
Fixed-point numbers are represented as 32-bit or 64-bit signed integers. The sign convention is two's-complement, bit 0 being the sign bit, and the binary point is assumed to be after the least significant bit.

B.3 LOGICAL FORMATS

Logical operations on 32-bit or 64-bit items treat them either as bit strings or as unsigned (that is, positive) fixed-point numbers.

B.4 DECIMAL FORMAT

Decimal numbers are held in a string of consecutive bytes, with two 4-bit, binary-coded decimal digits to each byte except for the least significant 4-bits, which correspond to a sign as follows



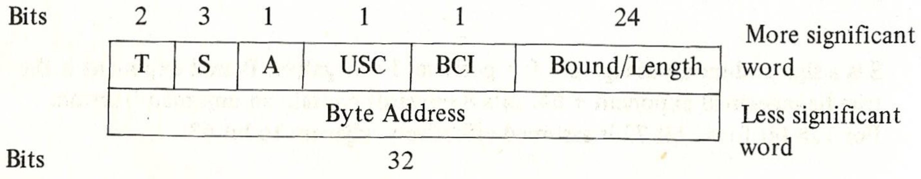
Recognised sign values are

1010, 1100, 1110, 1111	positive
1011, 1101	negative

Decimal numbers may thus be any odd number of digits but arithmetic is carried out on numbers of 7, 15 or 31 digits, plus sign.

B.5 DESCRIPTOR FORMATS

All descriptors are 64-bits. The format is as follows



The byte address may be modified in the course of accessing the information to which the descriptor occurs. The meaning of the other fields depends on the value of the type field, T, as follows.

T = 0 Vector Descriptor

S = size of addressed item(s) in store. Permitted values are

S	*Size in bits*	*(First) Item addressed*
0	1	First bit of byte addressed
3	8	Byte addressed
5	32	Word containing byte addressed
6	64	Word containing byte + next word
7	128	Word containing byte + next 3 words

A = ignored, should be 0. The values of USC are as follows

USC	= 1	unscaled, modifier is added directly to the address field
	= 0	scaled, modifier is scaled according to the size field, that is, it is taken to represent a number of items of the size denoted by S. If S = 0 this permits access to any bit within a byte.

The values of BCI are as follows

BCI	= 0	any modifier is checked (before scaling) to ensure that it is less than the value of the Bound field
	= 1	no Bound check.

Bound = unsigned integer upper bound for modifiers

T = 1 String Descriptor

S should always have value 3. A, USC, BCI are ignored (should be 0).

Length = length in bytes of string whose first byte is indicated by the address field (possibly after modification)

T = 2 Descriptor Descriptor

S must = 6, otherwise equivalent to T = 0.

T = 3 Code Descriptor

Fields S, A, USC and BCI taken together define a 6-bit subtype number, N. Permitted values are as follows

T = 3, N = 32 Bounded Procedure Descriptor. The address field references the destination instruction of a Call or Exit Instruction. Bit 63 is ignored as instructions must be half-word aligned. Modifiers (if any) are scaled by 2 (that is, represent half words). Bound is an upper bound for a modifier as with T = 0.

T = 3, N = 33 Unbounded Procedure Descriptor. As for N = 32 but the Bound field is ignored and no check is performed.

T = 3, N = 35 System Call Descriptor. The Bound field (usually) contains an entry displacement to index a System Call Index Table. The Address Field contains an entry displacement to index the System Call Table referenced by the descriptor accessed from the System Call Index Table.

T = 3, N = 37 Escape Descriptor. When a descriptor in DR being modified by a MODD instruction or used to access data indirectly is found to be of this type a branch is made to the address found in the word referenced by the Address field. This word will be followed by any parameters the escape routine needs. By using the Escape Exit instruction it is possible to return to execute the original instruction with a calculated descriptor.

References

1. J. K. Iliffe and J. G. Jolist, 'A Dynamic Storage Allocation Scheme', *Comput. J.*, 5 (1962) p. 249.

2. G. C. Scarrott and J. K. Iliffe, 'The Basic Language Project', in *Information Processing 68*, (North Holland, Amsterdam, 1968) p. 508.

3. J. K. Iliffe, *Basic Machine Principles* (MacDonald, London, 1968).

4. T. Kilburn, R. B. Payne and D. J. Howarth, 'The Atlas Supervisor', in *Programming Systems and Languages* (McGraw-Hill, New York, 1967) p. 176.

5. D. J. Howarth, 'A Re-appraisal of Certain Design Features of the Atlas 1 Supervisory System', in *Operating Systems Techniques* (Academic Press, London, 1972) p. 371.

6. *Operating Systems George 3 and George 4.* ICL Technical Publication 4267.

7. *B6500/B7500 Information Processing Systems* (Burroughs Corporation, Detroit, Michigan 48232).

8. E. I. Organick, *The Multics System: An Examination of Its Structure* (M.I.T. Press, Cambridge, Mass., 1972).

9. T. Kilburn, D. Morris, J. S. Rohl and F. H. Sumner, 'A System Design Proposal', in *Information Processing 68,* (North Holland, Amsterdam, 1968) p. 491.

10. P. C. Capon and I. R. Wilson. 'The Compiler Writer's MU 5',, *Conference on Compilers, Systems and Technology IERE Conference Proceedings* No. 25 (London, 1972).

11. N. Wirth, 'PL360, a Programming Language for 360 Computers', *J. Ass. comput. Mach.*, 15 (1969) p. 17.

12. D. E. Knuth, 'An Empirical Study of Fortran Programs', *Software-practice and Experience,* 1 (1971) p. 68.

Further details and up-to-date information on the 2900 Series can be obtained from

International Computers Limited
ICL House
Putney
London SW15 15W.

Index

Note Page numbers in italics refer to major sections.

The book is intended as an explanatory guide for readers who are familiar with general computing concepts and wish to gain a broad understanding of ICL's 2900 Series, its basic architectural features and its historical development.
In order to place the 2900 Series in context, the book begins by tracing its history and the internal and external influences on the architectural design. In particular, the aims and objectives for the new series which were decided at an early stage are enumerated and their influence on subsequent design described.
From this basis the 2900 Series is described at three levels of detail. The first level consists of fundamental concepts: abstract machine definitions of virtual machines, processes and program structure which impose a basic shape on a 2900 system. The second level describes the mechanisms which are used to support these concepts: the stack, descriptors, the protection system. The final level describes the primitive architecture: data formats, instruction types, interrupts.
The 2900 Series design is such as to allow a wide variety of actual implementation choices to exploit technological developments or to achieve a balance between cost and performance. Some concrete examples of early implementations are included at the end of the book.
Although the book is intended to be read sequentially, many of the individual sections are freestanding. It is therefore possible to use the book for various purposes, for example, to trace the effects of initially stated requirements on the final architecture, or to examine individual implementations as manifestations of architectural concepts.
John Buckle was a member of the original ICL architectural design team for the 2900 Series and later managed the design and implementation of the software tools used to implement 2900 systems. Before leaving ICL in 1975 to establish his own consultancy business, he was manager of the 2900 programme in the Product Development Group, responsible for the hardware and software development aspects of early 2900 deliveries. From 1966 to 1968 he was Professor of Computing Science at the Indian Institute of Technology, New Delhi, under the auspices of the Colombo Plan Technical Aid Scheme.