Chapter 17

An Outline of the ICL 2900 Series System Architecture

J. L. Keedy[1]

Summary The system architecture of the ICL 2900 Series is outlined informally. Its central feature, the virtual machine concept, is described and related to virtual storage, segmentation and paging. The procedural approach is then discussed and its implementation by a stack mechanism is described. Further sections outline the protection mechanisms, and the instruction set and related features. Finally the virtual machine approach is related to global system activities.

The paper has been written such that it may be of interest to readers without a specialist knowledge of computer architecture.

Shortly after its announcement in October, 1974, the ICL 2900 Series[2] was described in the popular computing press [Dorn, 1974] as little more than a copy of the B6700/7700 systems. It is easy to see how this happened, when one discovers that it is a stack oriented machine with a segmented virtual memory which makes extensive use of descriptors. In reality the implementation of these techniques is very different in the two computer families, and although a more serious attempt has been made to evaluate these differences [Doran, 1975] this is to some extent unsatisfactory since the author has, I believe, fallen into the same trap, albeit more subtly, of viewing the ICL 2900 through the eyes of someone thoroughly steeped in B6700 ideas. In fact, although the ICL 2900 has features in common with the B6700, radical differences exist, and some of the ICL 2900 features have more affinity to other systems, such as MULTICS [Organick, 1972]. Before the similarities and differences between such systems and the ICL 2900 Series can be fully appreciated, it is highly desirable that the ICL 2900 system architecture should first be understood in its own right. The real novelty of the architecture lies in the way in which its designers returned to first principles, and in the simplicity and elegance of the result. In this paper I shall therefore describe its architecture in a manner which attempts to reflect the thoughts of its designers, aiming at a level of description similar to Organick's description of the B6700 [Organick, 1973]. No attempt will be made to compare and contrast it with other systems, and it is hoped that the paper will provide an intelligible overview to readers without specialist knowledge of computer architecture.

1. The Virtual Machine

Faced with a problem to be solved using the computer, the user formulates a solution in a high level computer language such as COBOL or FORTRAN, and having satisfied himself of its correctness he will regard the resultant program as "complete." This is in one sense correct. His encoded algorithm will, if he has done his job well, be logically complete. However, even after it has been compiled, the user's program (or in more complex cases, his sequence of programs which comprise a job) must co-operate with other programmed subsystems (operating system, data management software, library routines, etc.) to solve the user's problem. The efficiency with which the problem is solved depends to a considerable extent on how the whole aggregate of necessary subsystems co-operates, and not merely on any one subsystem. It follows that it will be advantageous for a computer architecture to provide facilities for the efficient construction and execution of such aggregates. The 2900 Series explicitly recognises these aggregates, calling the environment in which each one operates a "virtual machine."[3] An aggregate itself is called a "process image," its execution by a processor is a "process," and its state of execution as characterised by processor registers is its "process state."

In the following sections we shall develop the idea of the virtual machine by considering its mainstore requirements, the dynamic relationship between its components, its protection requirements and its instruction set. But before we embark on this a few further remarks are necessary.

The fundamental concept, that each job runs in its own virtual machine containing all the code and data required to solve the application problem, allows the programmer to suppose that he is the sole user of the computer. But economic reality dictates that the real machine must be capable of solving several problems simultaneously, and this necessity for multiprogramming raises a set of problems which could threaten to destroy the advantages of the virtual machine approach. For example, how are the independent virtual machines co-ordinated, synchronised and scheduled? How, in view of high main storage costs, can separate process-images be permitted to have a private copy of common subsystems (e.g., the operating system)? How can virtual machines communicate with each other? Such questions will be borne in mind as we develop the concept of the virtual machine, and subsequently we shall consider them more directly, in an attempt

[1]*Australian Computer Journal*, vol. 9, no. 2, July 1977, pp. 53–62.

[2]References to the ICL 2900 Series in this paper are to the larger members of the new ICL range, which should not be confused with the ICL 2903 or the ICL 2904 computers.

[3]The term "virtual machine" has a wide variety of meanings in computer jargon. In this paper it is used consistently in the special ICL sense described here.

to show that the benefits and principles of the virtual machine are not compromised by the secondary modifications which are introduced to facilitate the efficient multiprogramming of several processes in separate virtual machines.

2. The Segmented Virtual Store

The relatively high cost of main store when compared with other storage devices, such as drums and discs, forces the computer architect to consider how this essential system component can be utilised with greatest efficiency. Amongst the more pressing problems in this area are:

a The process-image, and possibly even the user program alone, may exceed the size of available main store.

b Competition for main store by a number of programs may exist (e.g. in a time sharing system).

c Efficient use of main store for variable length tables, etc.

The most promising technique for solving such problems is the virtual storage concept, first used on the Atlas machines. In order to ensure that the user's needs are satisfied we shall look at this solution in the light of program structures.

The output of a compiler consists mainly of a series of logical regions comprising an object program. Most third generation architectures treat the object program as a single logical unit (e.g. for protection purposes), but certain advantages accrue if the logically separate regions, such as code sections and data areas, which we shall for the moment call program segments, are recognised as separate entities. For example, the separation of code segments from data segments considerably simplifies the production of "pure" reentrant code; this in principle allows separate virtual machines to use a single real copy of common code (e.g. operating system procedures) whilst allowing us to retain the concept of a process image containing all the code necessary to solve the user's problem. We shall see other advantages of the architectural recognition of segmentation in due course.

A characteristic feature of segments within a process-image is their need to cross-reference each other, the obvious technique for implementing this being to form an address consisting of segment number plus displacement within segment. If we now form for each virtual machine a "segment table" consisting of a list of entries (one per segment in the process-image), which map the segments onto main store addresses, and make this available to the hardware, then the hardware can calculate the exact main store location of any item cross-referenced by a "segment number plus displacement" address. If a segment table entry also contains a marker indicating whether the segment is present in main store, or is temporarily held on a secondary storage device (e.g. a drum), and a record of the length of each segment (see Fig. 1), then we have the rudiments of a segmented virtual store. This concept allows part or all of a process-image to reside temporarily outside main store on some secondary storage device, and thus in principle solves our problems of (*a*) a process-image which exceeds the size of main store, and (*b*) competition for main store usage in a time-sharing environment. Our remaining problem (*c*) of variable length segments can in principle be solved by allowing the recorded segment length to be changed.

The hardware procedure for translating a "segment number plus displacement" address (i.e. a virtual address) into a main store address is as follows. If P_i indicates that segment i is not in main store, the hardware causes an interrupt to allow the software to read the segment into main store; otherwise the virtual address i (segment number), j (displacement) is calculated as $R_i + j$. A further advantage of this scheme is that the test $j > L_i$ reveals erroneous attempts to jump to non-existent code or to access non-existent data beyond the upper bound of any segment.

Although this segmentation scheme is conceptually complete, the practicalities of multiprogramming require the introduction of certain modifications for the sake of efficiency. The existence of a separate entry in each virtual machine's segment table for those segments required in all virtual machines (e.g. operating systems

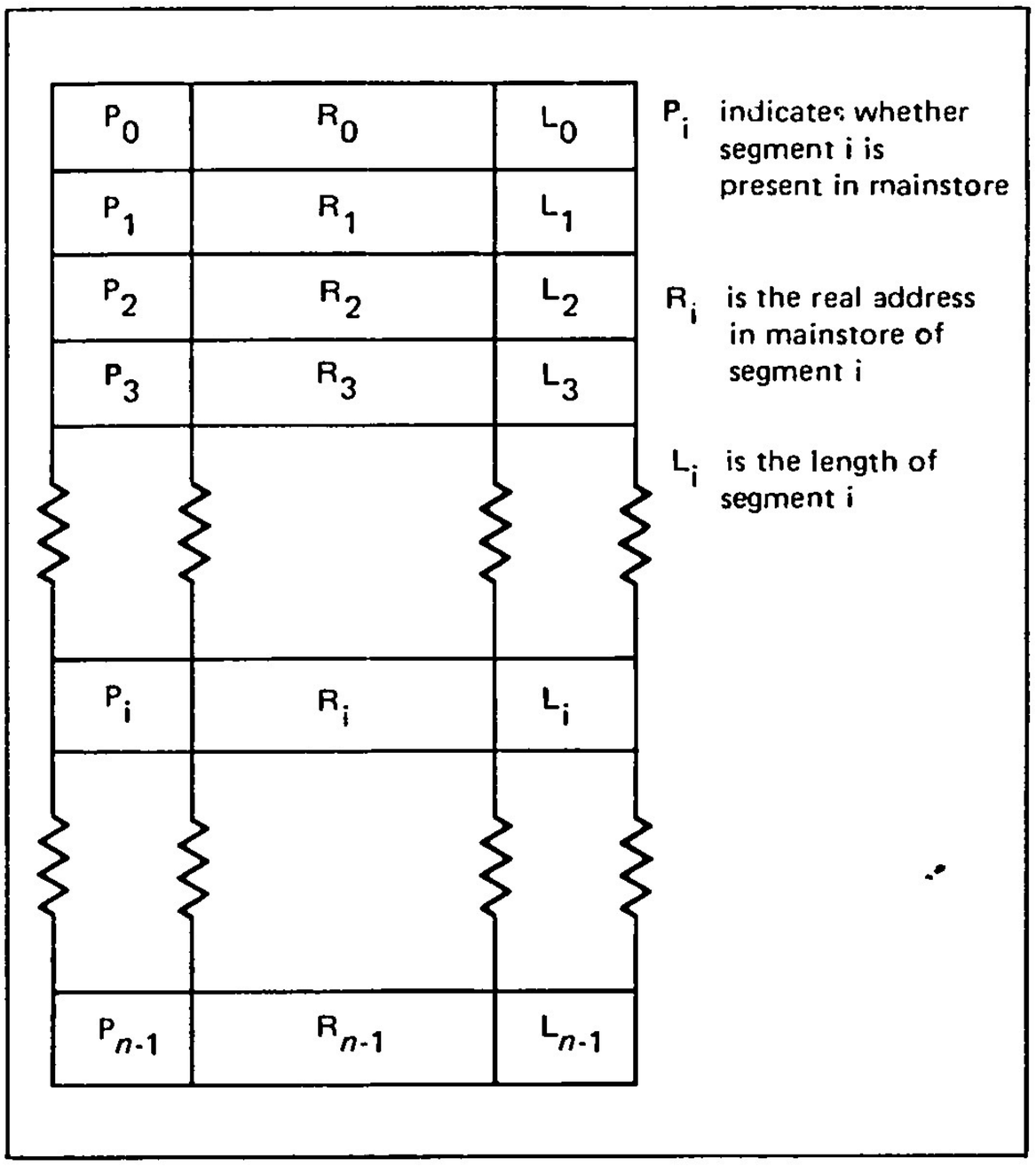

Fig. 1. A segment table for a process-image consisting of *n* segments

code) would not only be wasteful of space—it would also add significant overheads when moving such a segment around in the virtual store, since each segment table in the system would have to be updated. The solution adopted in the 2900 Series is to recognise a second category of segment table, the "public" segment table, containing an entry for each common or "public" segment. Since only one copy of this table need exist (thus saving space and allowing efficient movement of public segments) the process-image of a job is defined by a combination of its local segment table plus the one public segment table, and the hardware tests the most significant bit of a segment number to select the appropriate table (local segments are numbered 0-8191, public segments 8191-16383).

A third class of segment is shared locally between certain but not all virtual machines. Such segments, which are rather misleadingly called "global segments" are particularly useful for implementing real-time transaction processing on the 2900 Series. To implement such global segments as public segments has the undesirable side-effect that they would become accessible to virtual machines not privileged to access them, by virtue of their appearance in the public segment table. Since in practice global segments are relatively rare, to include them in each appropriate local segment table is unlikely to lead to a serious misuse of storage space, but the updating of multiple entries when the segment is moved, or when its length is changed, remains a difficulty (especially as each virtual machine sharing the segment may allocate to it a different segment number). The 2900 Series therefore permits a third class of segment table, the "global" segment table, which contains entries similar to other segment table entries. However, the global tables are not ordered by segment number, but are referenced via the local segment tables, which for global entries contain an indirection marker and in place of a segment's main store address the address of the appropriate global segment table entry (see Fig. 2). In this way movements of a global segment require that only the global segment table entry be updated, whilst rapid access is achieved via the local segment table.

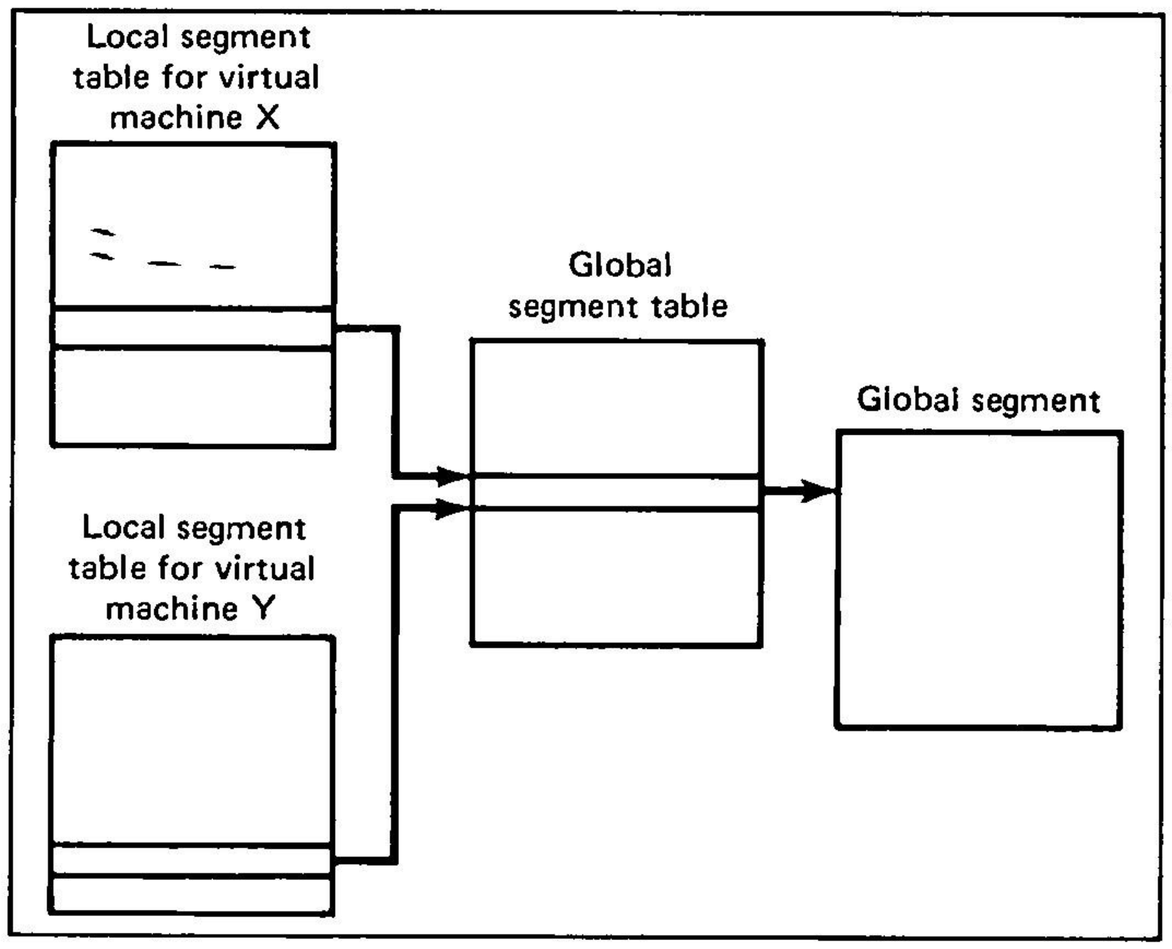

Fig. 2. Two virtual machines sharing a segment addressed via a Global Segment Table.

We now have an addressing structure capable of mapping virtual machines efficiently onto the storage hierarchy, but there remains the practical question of economic main store management. Since we have followed the most natural path by allowing variable length segments (with the additional potential space saving benefit of allowing the length of a segment to vary at execution time), we are forced to come to terms with the well-known problem of the "external" fragmentation of main store. This is illustrated in Fig. 3, which shows a map of a main store containing segments and holes left by segments no longer in main store; there is clearly enough free space for the new segment, but it cannot be loaded because the holes are not contiguous.

The 2900 Series designers examined the various solutions to this problem and decided to adopt the paging technique, whereby variable length segments are divided into fixed length pages, thus allowing main store allocation to be effected in fixed length blocks as is shown in Fig. 4.

This solution, which always allows a paged segment to be loaded provided that sufficient store blocks are free, requires the introduction of page tables, and the interpretation of a virtual address as "segment number plus page number plus page displacement." The actual 2900 Series virtual address structure is shown in Fig. 5, from which it can be seen that a virtual machine may contain up to 2^{14} segments each consisting of 2^{18} bytes divided into pages of length 2^{10} bytes.

The segment table entry is now modified to point to a page table (one per paged segment), which is indexed by the page number part of the virtual address and contains the main store addresses of

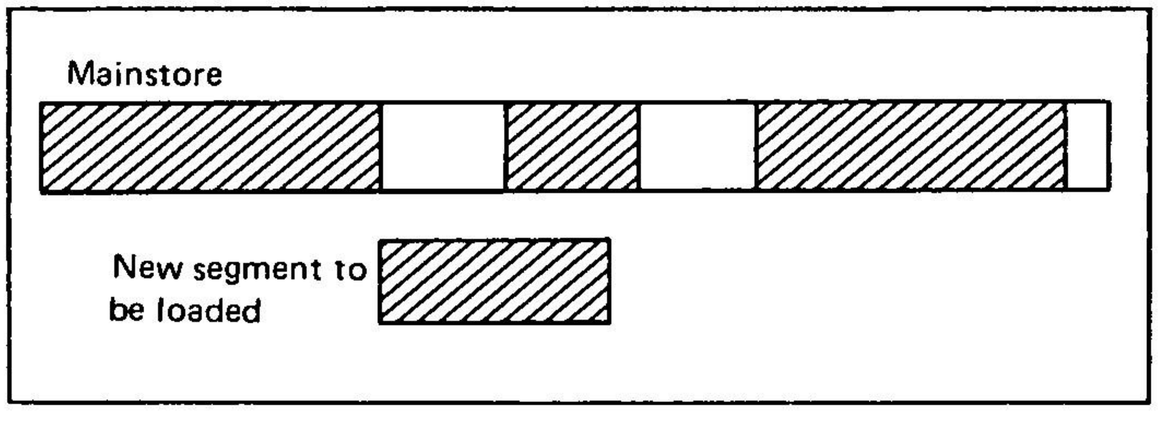

Fig. 3. An example of the external fragmentation of a main store.

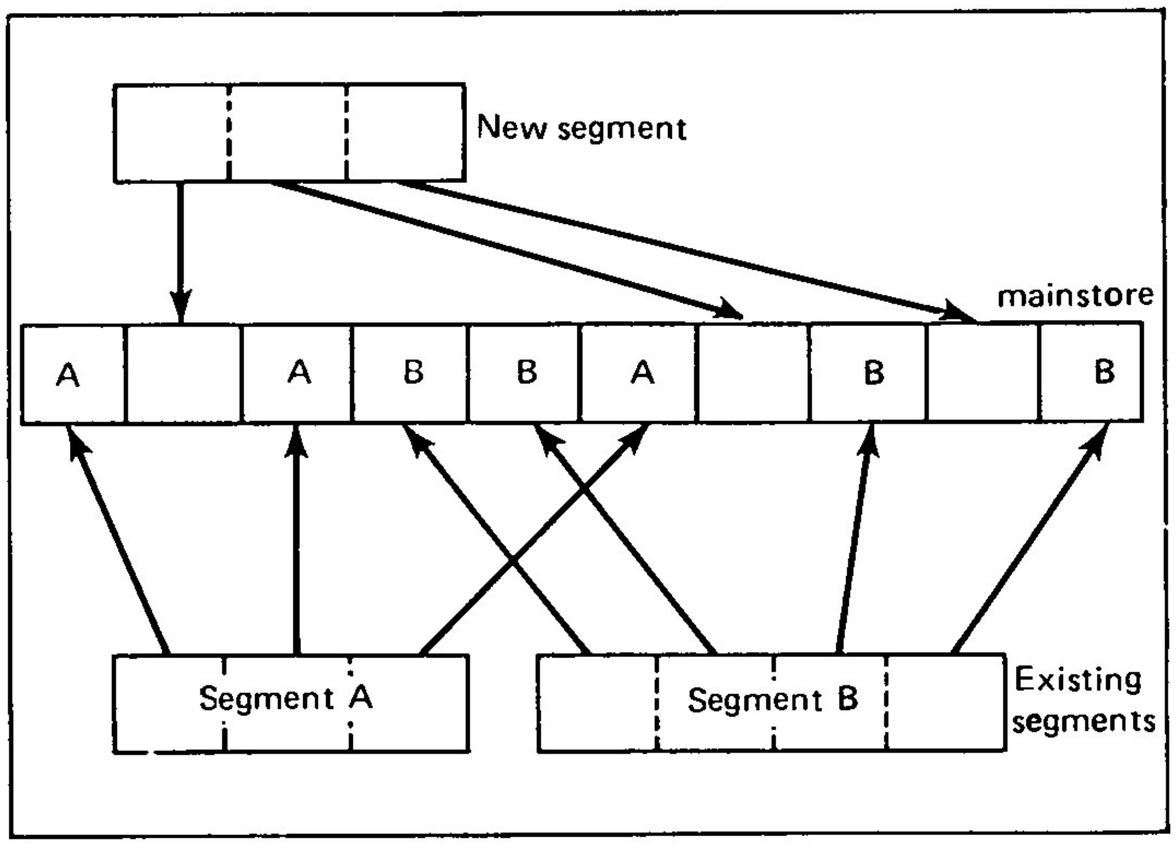

Fig. 4. Paged allocation of segments in main store.

all resident pages. Two of the advantageous side-effects of the paging solution are (*a*) that a segment can be brought into main store in stages, and (*b*) the length of segments can be extended without having to find a new block of store large enough to hold the whole segment.

The main drawback with paging is that it can lead to "internal fragmentation," the loss of main store space at the end of a segment caused by the necessity of rounding the segment length up to an integral number of pages. The average proportional loss of storage, assuming that the average segment size s is large in relation to the page size p, will be $p/2s$ (i.e. half a page per segment). It is intuitively obvious that this loss can be minimised in two possible ways: by keeping the page size small in relation to the average segment size and/or by attempting to produce segments whose lengths are as close as possible to an integral number of pages. But the page size must not become too small (otherwise the overheads of page tables and secondary store transfers become too great) and the segment size as previously defined is of an arbitrary length, dependent upon the logical program structure (and therefore not normally a multiple of page size). The 2900 Series therefore compromises by treating a physical segment as consisting of one or more logical regions, called areas, produced by a compiler. This gives the user the flexibility to create longer segments in relation to page size, and also to attempt to create segments whose length is as near as possible a multiple of the page size, and thereby to help reduce storage loss through internal fragmentation. For reasons which will become clear when we discuss protection the areas comprising a segment should share the same properties (e.g. read only data); and for obvious reasons only one variable length area can be included in a segment.

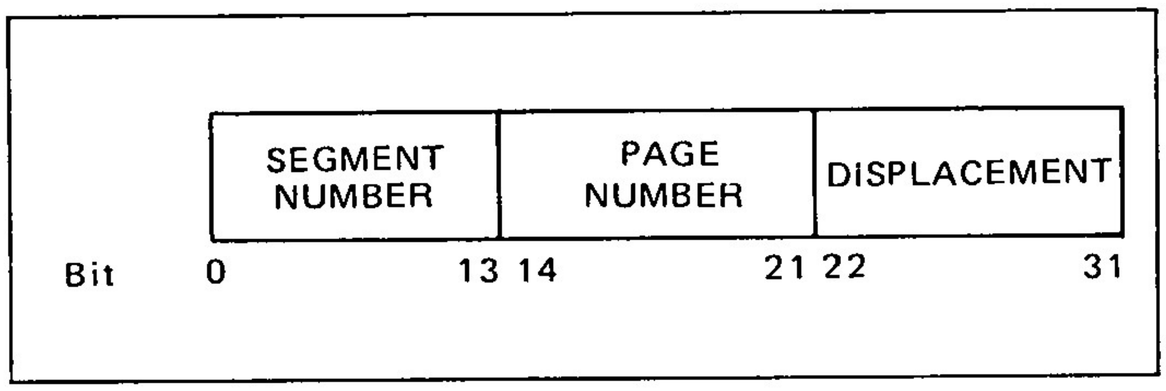

Fig. 5. The 2900 series virtual address.

Since there are some segments for which paging is irrelevant (e.g. main store resident segments of the operating system) the architecture allows for both paged and non-paged segments. Figure 6 shows the logical structure of the segment and page tables for a particular virtual machine.

3. Subroutines, Procedures and the Stack

We now return to the concept that the efficient execution of a user's task depends not merely on his own program but upon the

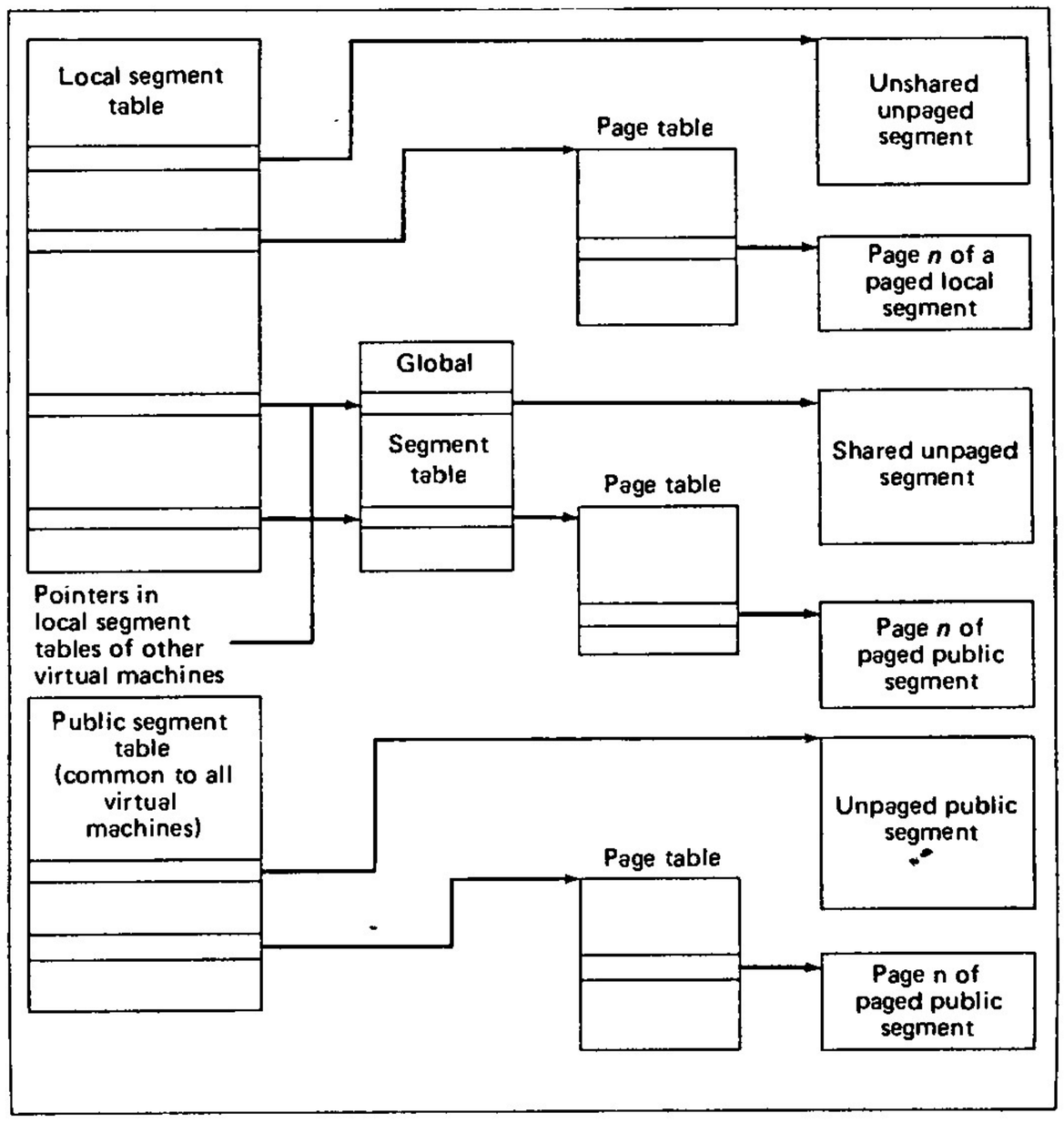

Fig. 6. The structure of the page and segment tables for a virtual machine.

totality of code and data required to solve his problem. It is essential that the virtual machine, the environment for executing such a process-image, provides mechanisms enabling efficient dynamic co-operation between the various subsystems comprising the process image.

Inter-subsystem calls are really only a special case of calls between code routines within the process-image, the more general case being the subroutine or procedure call, which appears in one form or another in all the major high level languages. The question may now be restated as: how can subroutine/procedure calls be flexibly and efficiently incorporated into the architectural model?

A relatively complex subroutine needs its own variables and work areas. If it is to be used recursively such work areas must be created on each entry to the subroutine. It also needs a mechanism for linkage with the calling code, which may also supply it with parameters. Such a subroutine is called in 2900 terminology a "procedure," and is implemented with the aid of a last-in first-out hardware assisted stack.

Each stack is held as a separate segment[1] and is controlled by four registers (see Fig. 7):

a Stack Segment Numbers (SSN)—the base address of the stack.

b Stack Front (SF)—the address of the next free location in the stack.

c Local Name Base (LNB)—the start address of the name-space for the current procedure or lexical level.

d Extra Name Base (XNB)—can be used for example to address the start of a previous lexical level in the stack or as an off-stack pointer.

[1]Thus a virtual machine may support several stacks, and therefore several (co-operating) processes.

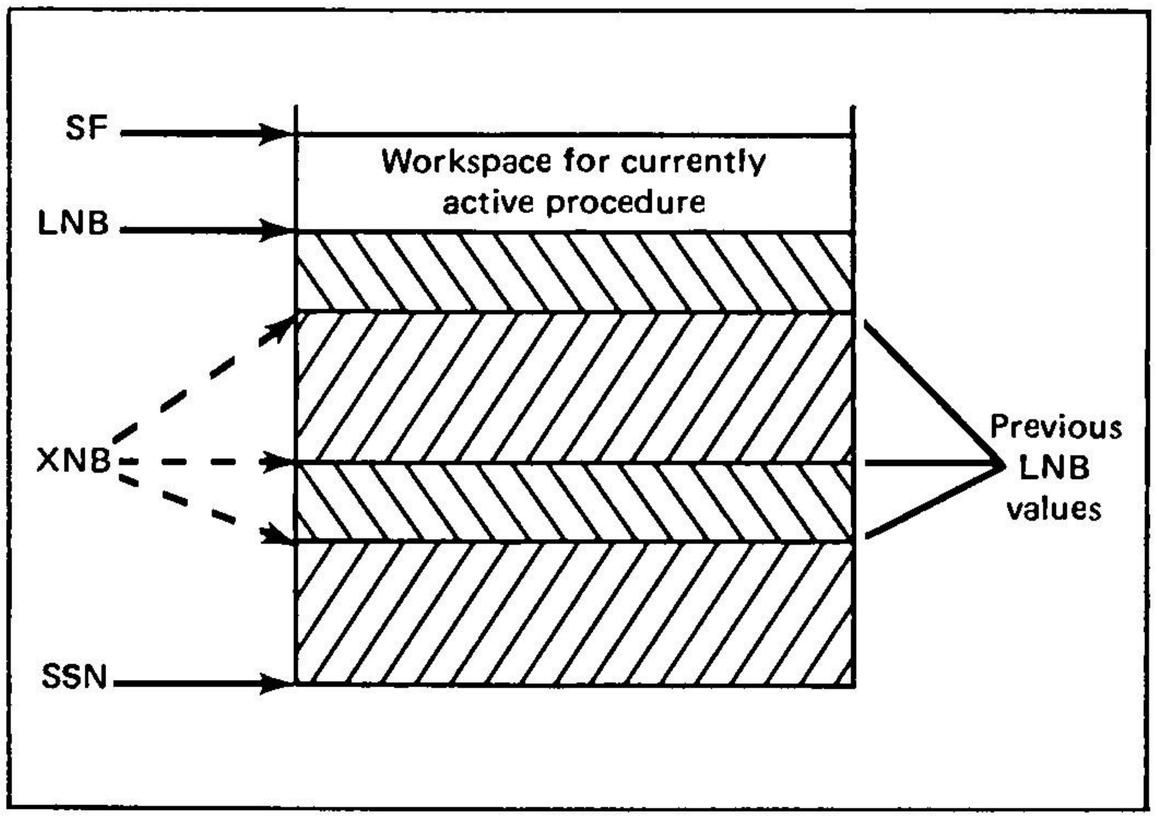

Fig. 7. Stack registers.

A procedure call takes place in two stages—a software pre-call sequence and a hardware call instruction. The software stores the current LNB value at the address held in SF, raises SF to leave space free for linkage data, stores the parameters at the new top of stack, and raises LNB to point to the next lexical level. The hardware call instruction then inserts the linkage data and in the normal case begins executing the new procedure (see Fig. 8). This procedure now has access to its parameters via LNB and to a new workspace starting at SF. It is free to call further procedures (or itself recursively) or to call the hardware exit instruction, which causes the stack to be collapsed back to the previous local name space, and the calling procedure will then be resumed at the instruction following the call instruction.[2]

4. Main Store Protection

One of the main functions of a computer architecture is to provide mechanisms which ensure that procedures have appropriate access to the data and code segments necessary for the execution of their task, but are not permitted to interfere with other segments in an unauthorised way. Such a requirement appears at two levels, within a virtual machine and between virtual machines.

Let us consider first the avoidance of interference within a single virtual machine. The most obvious example of the need for this is to prevent an untested user program from corrupting the other subsystems in its virtual machine.

The inadequacies of the traditional solution to this problem—the recognition of two classes of program (privileged software and unprivileged programs)—become obvious if we consider a "compile and go" system such as BASIC with the compiler itself

[2]There is, of course, a simple "jump and link"instruction (which stores a return address at the top of the stack) for use in implementing more trivial subroutines.

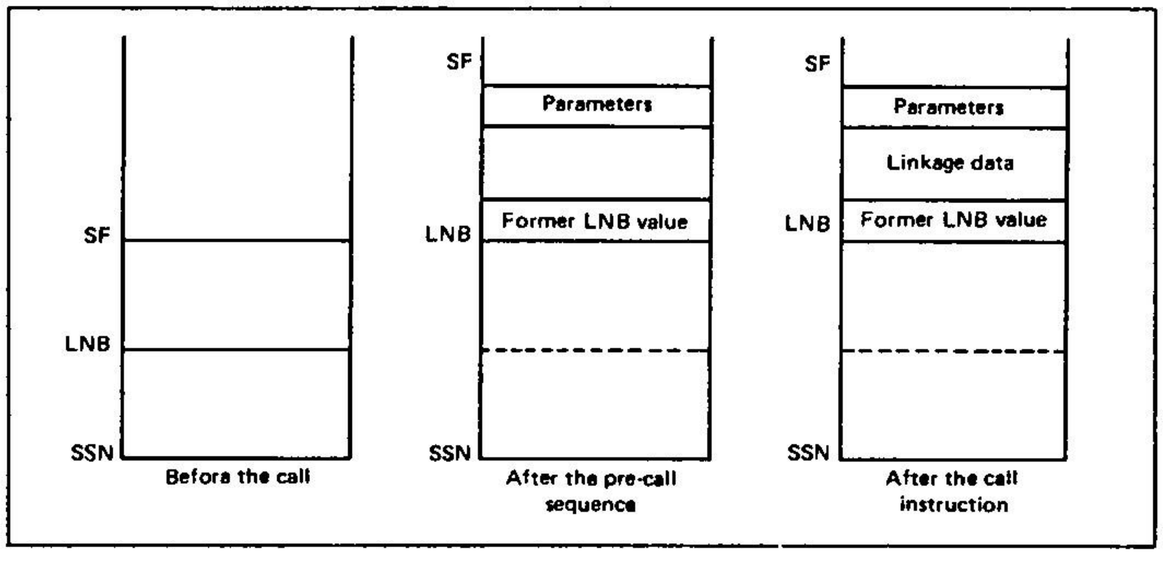

Fig. 8. The stages of a procedure call.

running as an unprivileged program: in this situation the compiler is unnecessarily endangered by the executing program. At the other extreme one could envisage a totally safe system in which each procedure has its own data class, and is only permitted to access data areas of this class; but the overheads in such a system would be high.

The intermediate solution selected for the 2900 Series architecture provides protection at the segment level (since this type of object is already known to the hardware) and associates with a process a 4 bit Access Control Register (ACR), allowing a range 0-15 of protection classes. Each segment table entry has an associated "read access key" (RAK) and "write access key" (WAK). Only if RAK $\geq$ ACR is the procedure permitted to read a segment, or if WAK $\geq$ ACR to write a segment. Likewise a segment can only be executed if a further marker in the segment table entry, the "execute permission bit" (EPB) is set.

The access control register is contained within the "program status register"(PSR), as is also a one-bit register known as PRIV, which in fact controls access to the PSR (and therefore to ACR). Under normal circumstances PRIV is reset, thus prohibiting changes to ACR (which reflects the protection level of the current procedure).

However if the procedure attempts to call another procedure which executes with a different ACR value, reference is made (either by hardware, or by a software interrupt routine running with PRIV on) to a software-created system call table to validate whether the call is permitted. If so ACR is assigned the value associated with the called procedure, PRIV is reset and the procedure is entered. Since PSR is stored on procedure entry as part of the linkage data, the hardware exit instruction can normally reload ACR with the appropriate value on returning to the calling procedure.[1]

By limiting the privilege of changing ACR to the lowest level of interrupt software access to segments within the same process image is properly controlled within sixteen levels of privilege—this being sufficient to provide a highly structured operating system with several levels remaining for user programs. The one apparent loophole in the scheme, the possibility that a less privileged level passes as a parameter to a more privileged level a manufactured but valid address to which it is not permitted access, is overcome by the provision of a special hardware "validate" instruction, which the called routine uses to find out what type of access (if any) the ACR level of the calling routine has to data at the address supplied. This is possible because the linkage information in the stack contains the ACR value of the calling routine, which can be checked against the WAK and RAK values in the segment table entry for the address to be validated.

For obvious reasons the architecture must also provide a mechanism to ensure that certain instructions (e.g. instructions controlling input-output devices) are not misused, and this is also achieved by testing the PRIV bit in the PSR.

Finally, the question of store protection between virtual machines (i.e. prohibiting interference between user jobs) is automatically solved by the addressing structure. Any address used within a virtual machine is transformed into a real address by means of the virtual machine's own segment tables. It is simply impossible to access a location not contained in these segment tables.

5. The Instruction Set

The instruction set for the 2900 Series was designed specifically with the needs of high-level languages in mind, and its objectives include efficiency of compilation and execution, reliability of execution, and compactness of object code. In order to achieve these objectives the 2900 Series instruction set interlocks closely with descriptors, registers and the stack in manipulating the basic data formats.[2]

Although the segmentation protection scheme provides a fair degree of execution reliability (e.g. by ensuring that data is not "executed" as code, that code and read-only data cannot be corrupted, etc.), this is oriented to ensuring non-interference between different subsystems and programs. In order to provide a means of detecting execution errors within a single high-level language program or subsystem (e.g. an attempt to access an array element beyond the array boundaries, or to perform an indexed jump beyond the boundaries of a specific code module) the 2900 Series employs "descriptors," which associate with a virtual address a description of the object addressed.[3] Descriptors, which provide other facilities in addition to run-time error checking, are in four standard formats each consisting of 32 bits for the description plus 32 bits for the address:

[1]This description refers to calls which reduce ACR value (i.e. increase privilege), and to the corresponding returns. Calls which increase ACR value (i.e. reduce privilege) and corresponding returns involve the creation of an additional stack to ensure that on-stack data is not available to non-privileged code; such calls are generally avoided, because of the overheads involved in creating a new stack.

[2]Bits; 8-bit bytes in EBCDIC and packed decimal formats; 32 or 64 bit words containing logical or fixed-point numerical values; 32, 64 or 128 bit words for floating-point numbers; 32, 64 or 128 bit words containing 7, 15, or 31 digit signed decimal integers.

[3]Unlike MULTICS or B6700 descriptors, the ICL 2900 descriptor mechanism is internal to an address space, rather than a means of defining the address space.

Vector Descriptors contain a size field indicating whether a data element is 1, 8, 32, 64 or 128 bits in length; a bound field containing a count of elements; a bound-check inhibit indicator; and a scale bit which indicates whether address modifiers are to be scaled in accordance with the size field. The vector descriptor can be used to address individual primitive data items (such as an integer variable) or single dimension arrays of primitive elements. Provision for multi-dimensional arrays is in the form of dope-vectors consisting of triplets, each describing a dimension.

String Descriptors describes rows of bytes, e.g. character strings, and hold an indication of the string length.

Descriptor Descriptors point to other descriptors and thus provide an indirect addressing facility.

Code Descriptors consist of normal code descriptors, system call descriptors, and escape descriptors. Normal code descriptors serve as operands for procedure call instructions not requiring a change of privilege, and for exit instructions not requiring an increase in privilege. System call descriptors contain instead of an address a pair of indices which reference a System Call Table entry (see Sec. 4); they are used as operands for procedure calls requiring a change of privilege and for exits requiring an increase of privilege. Escape descriptors, however, may be interchanged with any other descriptor as an exceptional means of by-passing normal code sequencing rules. On detection of an escape descriptor as an instruction operand the hardware causes entry to the code routine whose address is held in the escape descriptor, without executing the instruction for which it serves as an operand. The escape routine might typically monitor the use of a particular table or procedure, or instigate the loading into virtual store of some exception procedure not normally required (e.g. an error routine). A special mechanism is available to allow an escape routine, having placed the "correct" descriptor in a register, to cause the original instruction to be executed "correctly," and thence return to the normal code sequence.

In considering what form the register set should take, it was evident almost from the beginning that special purpose registers with dedicated functions would be more suitable than interchangeable general purpose registers for a "high level language machine" such as the 2900 Series. The problem with the latter is that compiler-writers, not being in a position to predict in advance the dynamic execution of programs to be compiled, are forced into following a set of conventions, which may be wasteful but which certainly distorts any theoretical advantages of having general purpose registers. On the other hand dedicated registers, if carefully designed, provide an appropriate tool for the compiler writer. At the same time they allow the hardware designer to optimise his implementation on the basis of the known purposes of the registers.

The 2900 Series provides the compiler-writer with both a dedicated set of registers and a virtually infinite number of on-stack locations which in practice serve as registers.[1] In addition to the four stack registers already described (SSN, SF, LNB, XNB) the following registers are "visible" to each process-image in unprivileged mode: a variable length accumulator (ACC) whose size (32, 64 or 128 bits) is controlled by a 2-bit register ACS; an index modifier (B) used mainly for address modification; a descriptor register (DR) used for addressing operands; a program counter (PC); a real-time clock (RTC); an overflow indicator (OV); a condition code (CC); and a program mask (PM) used to inhibit specific program interrupts.[2]

There are 113 functions in the instruction set, providing facilities for arithmetic, character manipulation, logical operations, instruction sequencing, etc. Most instructions operate on two operands, one of which (normally a register holding data or a descriptor) is usually implied by the function. The other operand may be a string whose descriptor is held in DR or may be specified in the operand field of the instruction (e.g. as a literal, a displacement from a stack register, etc.). The size of an instruction is 16 or 32 bits, depending on the method of specifying the operand rather than on the function code. Similarly function codes are not dependent on the length of the accumulator, so that the same functions are used, for example, in single precision and double precision floating point operations. To illustrate how the interplay of descriptors, registers and the stack results in efficient and compact object code which can be efficiently compiled from high level languages, we take three brief examples: arithmetic, array handling and character spring manipulation.

The stack is of course particularly well suited to the evaluation of arithmetic expressions by means of "reverse Polish" notation, and typical sequences such as "store accumulator value at top of stack, raise top of stack pointer, load new value into accumulator," or "add (multiply etc.) accumulator value and top of stack value, lower top of stack pointer" are efficiently compacted into single instructions.

Array handling will typically consist of a logical subscript value

[1]Since the more powerful machines have a slave-store dedicated to the stack, this statement is true not only logically but also in terms of physical speed.

[2]In reality ACS, OV, CC and PM are visible parts of the invisible (privileged) register PSR (program status register) which also contains ACR and PRIV. Other invisible registers include SSR (system status), LSTB and PSTB (base registers for the local and public segment tables), an interval timer (IT) and an instruction counter (IC). The totality of registers is called the "image store."

held in the B register operating on an array addressed by a vector descriptor. A single hardware instruction is able to check that the subscript does not exceed the bound of the array, and to find the start address of the logical element required by scaling the subscript (using the size field in the descriptor). Special functions also exist for performing efficient index arithmetic on multi-dimensional arrays.

Since the 24-bit length field of a string descriptor (rather than the instruction itself) can determine the length of store-to-store operations for character manipulation, long operations can always be performed as single instructions[1], and need not be broken down into sequences of shorter operations (say 256 bytes in length), as on some machines.

These examples not only illustrate the tendency in the 2900 Series instruction set to efficiency and compactness of the object code produced, but point also to simplifications in the compilation phase by reducing the necessity for performing arbitrary tasks such as top of stack pointer manipulation, subscript scaling and character string length checkup.

6. The System as a Collection of Virtual Machines

Whilst not ignoring the problems raised by multiprogramming, the previous discussion has looked at the architecture largely from the view point of a single virtual machine. The emphasis now changes as we consider such questions as: How is the allocation of real resources (e.g. processor time, mainstore) to virtual machines controlled? How are external interrupts (e.g. peripheral interrupts) handled? How is the use of shared data segments synchronised? In other words, how can virtual machines be forced to co-operate with each other?

The answer must be: these tasks are carried out by one or more subsystems. We shall call them collectively the "Kernel." But how can the Kernel be integrated into the architectural model described above without distorting it beyond recognition?

One possibility would be to provide the Kernel with its own special virtual machine—an apparently attractive solution if we take external interrupts into account. But since the data necessary for handling interrupts (e.g. a peripheral request table) originates from procedure calls in other virtual machines, the solution in fact implies radical modifications to the architecture.

The alternative solution, to consider the Kernel as a component part of every virtual machine, also requires modifications but these are more in the spirit of our fundamental principles. The Kernel will itself be held in public code segments in order that it can operate in any virtual machine, and will make use of public data segments to store information relating to such functions as scheduling. Some mechanism for synchronising the use of these shared data areas will of course be required to maintain the integrity of the data, and since the model has not placed restrictions on the use of public and global data segments this synchronisation problem can in fact arise in subsystems outside the Kernel. For this reason and because it does not solve the problem for a system with multiple processors, the use of non-interruptible code execution does not adequately solve the problem of synchronisation. The 2900 Series designers therefore included a variant of the semaphore solution [Dijkstra, 1968a].

The semaphore is an integer associated with a resource to ensure that it is allocated exclusively to one process at a time, and takes the values:

−1	Resource free
0	Resource in use—no waiting processes
1	Resource in use—one waiting process
n(>0)	Resource in use—n waiting processes

Assuming that processes co-operate by accessing shared tables, etc. via a semaphore (it is to their advantage to do so), then the mutual exclusion problem is limited to testing and updating the semaphore itself; this is solved by providing two non-interruptible hardware instructions—"increment and test" (which adds one to the semaphore and sets a condition code indicating its new status)—and "test and decrement" (which sets a condition code showing the status of the semaphore then decrements it by one).

"Increment and test" allows a process to request use of the semaphored resource and test whether the request was successful (condition code zero) or whether the process was merely added to the count of waiting processes (condition code positive).

"Test and decrement" allows a process to relinquish a semaphored resource which has been allocated to it, and to test whether the other processes are waiting (condition code positive). The missing link in this scheme, the ability of a process relinquishing the resource to advise a waiting process, is supplied by an event system which permits waiting processes to suspend on an "event" and relinquishing processes to cause the event.

Control of the event system is of course a function of the Kernel, and the scheme can be used independently of semaphores, to provide a general purpose synchronising facility. For example a user program can associate an event with a peripheral access request, and so be informed by event of the request termination. A process may cause an event, test for occurrence of an event, suspend on events, or nominate an interrupt routine to be entered on the occurrence of an event. The flexibility of the

[1]Such operations can be interrupted by hardware and subsequently resumed, thus ensuring that time critical interrupts are not delayed, and that virtual store interrupts arising from non-presence in main memory of (part of) one or both operands, can be serviced in mid-instruction.

event system is further improved by the provision of a primitive message passing facility (e.g. an indication of the success or failure of the peripheral request), thus creating a powerful mechanism for virtual machine synchronisation and communication.

There remains now the question of interrupt handling by the Kernel. Since we have defined the Kernel as a component of all process-images, it is evident that external interrupts will be accepted, and the initial decoding performed, in the currently active virtual machine. An attractive implementation of this is to treat interrupts as forced procedure calls, thus automatically storing the interrupted process state in the stack and at the same time creating a new working space for the interrupt routine. Unfortunately this solution runs into difficulties with interrupts whose purpose is to signify that there is no more space in main store at the top of the stack. Thus virtual store interrupts (and all interrupts of higher priority) are directed to a special stack known to the hardware, which, however, operates in all other respects like a normal stack.

Conclusion

The features of the 2900 Series system architecture described in this article are not peculiar to a particular model within the 2900 Series, but provide the basis at an architectural level for a compatible range of models, varying considerably in power and cost. This is achieved by means of two interfaces—the "Kernel Interface" which embodies the general architectural model, and the "Primitive Level Interface" which defines the instruction set and associated features. Neither of these interfaces can be regarded as a purely hardware interface, since the cost and power objectives of a particular model in the range will determine what is economic to implement as hardware, what as microprograms, what as software, etc.

The Kernel cannot be regarded as an operating system—it does not even provide a logical facility for communication between the operator and either the system or user programs—but is rather a primitive layer of software which provides further levels of software (operating systems, data management systems, etc.) with a consistent abstraction of the architectural model, regardless of the implementation details of individual computers in the range. Thus the Kernel Interface guarantees to the higher levels of software that resources (whether hardware resources such as peripheral channels or software resources such as events) are handled in a uniform manner and within the virtual machine framework provided by the lower level.

The Primitive Level Interface corresponds approximately to a hardware instruction set, but like the Kernel Interface, its description does not imply its mode of implementation. Thus it is to be expected that for smaller models in the range some functions (e.g. floating point operations) might be implemented in the Kernel software. Similarly whilst the larger models will use special rapid storage locations to implement registers, at the lower end registers might be implemented in ordinary main store locations. The importance of these two interfaces is that taken together they create an abstract machine which provides an efficient and reliable environment for the compilation and execution of user programs written in high level languages.

References

Dijsktra [1968a]; Doran [1975]; Dorn [1974]; Keedy [1976]; Organick [1972]; Organick [1973].