The hardware/software interface of the ICL 2900 range of computers

D. H. R. Huxtable and J. M. M. Pinkerton

International Computers Limited, Lovelace Road, Bracknell, Berkshire RG12 4SN

The paper aims to outline those ICL 2900 hardware features that support system software. The treatment broadly is in terms of the instruction set and other hardware features. 2900 architecture provides for concurrent execution of independent processes in virtual machines, so that the processes are protected from mutual interference. The protection arrangements are outlined together with ways of addressing the store, for which several kinds of descriptors may be used. There are both visible and invisible hardware registers; access to the latter is privileged via a so called image store addressing mechanism. The arrangements for handling interrupts and entering procedures are outlined. Slave stores are provided for reducing store access demands for frequently accessed items. Peripheral transfers are via autonomous and asynchronous controllers activated by processes operating within a central processor. In conclusion some of the main considerations influencing design decisions are reviewed.

(Received April 1977)

1. Introduction

The ICL 2900 range was designed with both the hardware and the operating system software in mind; that is, with a view to presenting an overall system to the user. This paper contains an account of the underlying architecture with special reference to those features of the instruction set that are designed to support the software. It is presented as a hardware description, but it is really a description of an interface and the implementation in actual hardware of the features described may vary from one model in the range to another: for example some registers may in one model be composed of bistables and in another be simply locations in main memory that are used by the hardware for the purpose.

At the primitive level the hardware supports:

1. A virtual memory with segmentation and paging.
2. A process stack.
3. Memory protection with a series of levels of increasing privilege and with special features to permit the running of diagnostic programs during normal operations.
4. A comprehensive interrupt and fault handling system closely integrated with the procedure call mechanism.
5. Slave (buffer) stores made safe in relation to channel transfers and multiprocessor operation by a consistent system of hardware provision and software convention.

The above hardware interface is designed to support an operating system that is stack oriented and provides each user with a virtual machine of his own. The writer of subsystems on which the ultimate high level user depends has available to him, insofar as he requires them, similar hardware support features to those that are available to the writer of the operating system itself.

2. The process stack and storage access

Virtual addresses contain 32 bits of which 14 define the number of the segment. The remaining bits are divided into page number, word number, and byte number. In order to address whole words only the first thirty bits are necessary.

A number of the registers listed in Appendix 1 serve the purpose of supporting a stack of which there is one associated with each process. The *stack segment number* register (SSN) is loaded by the operating system—specifically by the ACTIVATE instruction—with the segment number of the stack when a process is activated. SSN contains 14 bits and when concatenated with 16 zeros gives the virtual address of the base of the stack. The first 14 bits of the *stack front register* (SF) and the *local name base register* (LNB) are identical with those of SSN and these registers therefore contain pointers into the stack segment. SF, as its name implies, points to the stack front (more exactly to the next available location) while LNB always points to somewhere within the stack so that LNB < SF. The convention adopted in this paper is that the abbreviation may stand either for the register or its content. Conventionally LNB is used by the operating system to point to the working area of the current procedure; when a new procedure is called the old value of LNB is recorded at the top of the stack and LNB updated so as to point to the working area of the new procedure. Instructions that manipulate the stack by placing items on it, by removing items from it, or by making an explicit adjustment to SF carry with them hardware checks that lead to an interrupt if the condition LNB < SF is violated.

The instruction set permits direct access to be made to items on the stack by specifying their addresses relative to LNB. Only locations at or above that pointed to on LNB can be addressed in this way and the hardware checks that they are below SF.

In addition to LNB there are two other registers, the *extra name base register* (XNB) and the *cross-reference table base register* (CTB). These are of the full 30 bits and while they may be used to point in to the stack they are not generally so used. XNB may sometimes be used for keeping track of a former name space when a new one has been created by a procedure entry. Direct access to the store may be made relative to XNB or CTB in exactly the same way as reference is made relative to LNB. Since these references are not necessarily to the stack there can be no hardware checking at this point. However, the checking facilities provided by the virtual storage accessing mechanism for ensuring that accesses do not go outside segment limits are available.

The hardware is also capable of making access to the store via a *vector descriptor*. Vector descriptors consist of two words and specify a consecutive set of items in the store. The items must all be of the same size which can be one bit, one byte, one word, two words, or four words. Three bits in the first word of the descriptor give the size and other bits give the length of the vector, that is, the number of items that it contains. The second word contains the virtual address of the base of the vector. The *accumulator* (ACC) is four words long, but it may be set to operate if desired at the shorter lengths of two words or one word. The combined effect of these facilities is that strong support is given for single, double and quadruple length working.

Access to an item via a vector descriptor may be *direct* or *indirect*. For the former kind of access to be possible there must be a copy of the descriptor in the *descriptor register* (DR), it having either been placed there by an explicit instruction or left there as the result of a previous operation. The vector may then be accessed using as an index (offset) either a literal in the instruction or the content of a storage location specified relative to one of the following: LNB, XNB, CTB, *program counter* (PC). As an alternative, an *indirect access* may be made via a descriptor contained in the store. The location containing the descriptor is specified relative to one of the registers just mentioned. The first stage of the operation is to transfer this descriptor to DR; a second access is then performed, the offset, if any, being given by the number in the *Index Accumulator* (B).

A modifier used to give an offset to the base address given in a descriptor is first checked against the length field and an interrupt caused to occur if the check fails. The modifier is then scaled according to the size given in the size field and added to the base address. Checking and scaling will, however, not take place if inhibit bits in the first word of the descriptor are set.

More detail about the various ways in which the store may be addressed will be found in Appendix 2. In addition to vector descriptors there are string descriptors, descriptor descriptors, code descriptors, system call descriptors, and escape descriptors. These are all distinguishable by certain combinations of bits in the first word.

3. Access to the image store

It is a feature of the design that all hardware registers should be addressable by suitably privileged programs. To this end the hardware registers are conceptually grouped together to form what is known as the *image store*, in which each register has its own address. Some registers are only accessible in this way and are known as *invisible* registers. Others take part in operations called for by nonprivileged instructions available to the ordinary programmer. These are known as *visible* registers, the most obvious example being the accumulator. The functions of the most important registers in the image store will be explained below. Appendix 1 contains a reference list of registers in the image store, together with the abbreviations that stand for them.

The image store gives the appearance of being a consecutively addressed set of 32-bit locations. Image store locations may be addressed as such by using appropriate instructions from the regular instruction set, with image store addresses formed in the way described in Appendix 2. Such instructions count as privileged instructions. Some image store locations are by their nature read-only; for example, that corresponding to the real time clock. Locations corresponding to the visible registers are read-only, since these registers can be written into by using unprivileged machine instructions.

Access to the image store is controlled by two bistables known as PRIV and ISR. If PRIV = 1 (that is if the bistable is set) both read and write access—where the latter is possible—is allowed. When PRIV = 0 no writing access is permitted, subject to an exception concerning diagnostic programs that will be explained later. If ISR = 1 read-only access to the image store is permitted.

PRIV is itself represented by a bit in one of the image registers namely the *program status register* (PSR); other bits in this register serve such purposes as indicating whether arithmetic overflow has occurred, what the current size of the accumulator is, and so on.

ISR is represented by a bit in the *system status register* (SSR); other bits in this register identify the type of processor in use (this information is wired in and therefore unalterable), mask interrupts, control a real addressing mode that permits the segment tables to be bypassed when an initial load is being performed, and serve other similar purposes.

It will be seen that PRIV and SSR, being themselves bits in invisible registers, can be changed by program only when PRIV = 1. PRIV becomes set equal to 1 automatically when an interrupt occurs or when an explicit system call is made, the latter being equivalent to an interrupt. The system routine entered after an interrupt thus enjoys complete privilege. Having serviced the interrupt it will make use of an ACTIVATE instruction to restart the interrupted process or perhaps to restart some other previously suspended process. One of the functions of the ACTIVATE instruction is to set PSR and SSR so as to give the activated process its correct degree of privilege. Thus ACTIVATE can reduce privilege but not increase it; the same applies to an EXIT instruction used to return from a procedure.

It is desirable in the operation of a large computer system to be able to test peripheral devices and the circuits associated with them while the system is running a normal load. This is done by inserting a diagnostic program which runs as a job under the system. So that it should be able to perform adequate checks, this program must be able to write into buffer registers and other invisible registers associated with the device that it is testing. With the system described above it would be necessary for the diagnostic program to be given complete privilege and this is clearly undesirable. Accordingly it is arranged that access to registers needed for diagnostic purposes can be obtained if another bit known as DGW in SSR is set, provided that a diagnostic allow bit associated with the register or registers concerned is also set. This system enables diagnostic programs to be run with a minimum of danger to system integrity.

4. Virtual store addressing and protection

Implementation of the virtual memory makes use of segment tables and page tables in a manner that is now familiar. The base of the segment table corresponding to the currently operating process is held in an invisible register known as the *local segment table base register* (LSTB). There is a second invisible register, the *public segment table base register* (PSTB). Local segments—local, that is, to a process, although they may be shared with other processes—are those in the range 0 to 8191 and public segments are those in the range 8192 to 16383. Thus, whenever a segment is being accessed, the hardware refers to LSTB or PSTB according to the value of the most significant digit in the segment number. In 2900 terminology a local segment table is said to define a *virtual machine*; a number of virtual machines may share the same public segment table. Within a virtual machine each process has its own stack.

Each segment table entry consists of two words which, in addition to the effective address of the segment and the segment size, contain information about whether the segment is paged or not paged and whether it is shared or not shared. If the segment is shared the effective address points to the relevant entry in a *global segment table*. If the segment is not shared, the effective address points either to the base of the segment if it is not paged or to the base of the relevant page table if it is paged. In order to provide information to the store management system the hardware sets bits in the page tables of paged segments to indicate whether pages have been read from or written into. In the case of unpaged and unshared segments similar bits are set in the segment table.

A nine bit field in the segment table entry is known as the *access protection field* (APF) and is used for virtual store protection. Four of these bits constitute the *read access lock* (t_1), four constitute the *write access lock* (t_2), and one is the execute permission bit (t_3). When access to a segment is attempted a check is made that t_1 > ACR or t_2 > ACR as the case may be, ACR being a register containing the access key for the domain of a process in execution. Failure of this check leads to an interrupt. A segment may not be executed as code unless $t_3 = 1$.

Exceptions to the above rules are made in the case of read or write accesses made by a process to its own stack. A local segment may thus be given very low values of t_1 and t_2 and thus be protected from possible corruption by procedures running at higher levels, while yet remaining accessible to a procedure, running at similar levels, that uses it for its stack. This does not lead to any loss of protection, since the stack segment is protected by the way in which it is generated and also by certain checks that take place on stack register operations. The stack is, in fact, largely manipulated by hardware mechanisms.

5. The interrupt mechanism

Interrupts in the 2900 range are classified as asynchronous or synchronous. Asynchronous interrupts come from outside, typically from a mechanical peripheral or from a clock. Synchronous interrupts, which in other systems are often referred to as traps, occur during the operation of an instruction. They can arise either from some unexpected happening such as an accumulator overflow or a program error, or they can be deliberately provoked as, for example, in a CALL instruction, when it is necessary for the operating system to be entered at a high level of privilege to adjudicate on some matter of protection. Thus, the 2900 range handles all program changes that require the attention of the operating system in a uniform manner whether they arise as a result of an interrupt or are explicitly called for by the programmer.

Synchronous interrupts take effect as soon as they occur, a bit in SSR being set if the current instruction is incomplete. Asynchronous interrupts are dealt with according to an order of priority depending on the class to which they belong. Various internal interrupts, of which accumulator overflow is typical, may be masked by setting bits in PSR. System interrupts, including virtual store interrupts, may be masked by setting bits in SSR.

The *event pending* bit (EP) in SSR enables the system programmer to arrange that a delayed interrupt shall occur, for example, on emergence from a system procedure. If the EP bit is set, such an interrupt occurs on the execution of any EXIT instruction that causes ACR to be increased. The interrupt is masked however, if the appropriate bit in SSR is set. An interrupt can also occur in certain circumstances if the EP bit is set when a process is reactivated by an ACTIVATE instruction.

On receiving an interrupt, the hardware brings about a forced entry into a procedure designed to service the interrupt, first storing the process and program status on the stack so that it can be restored on return. Some interrupts can be processed using the same stack as the interrupted program; calls to the operating system are included under this heading. Others, such as external interrupts and virtual store interrupts, require that a temporary switch to another stack should take place. Interrupts are divided into classes and for each class there is an entry in a table known as the *interrupt steering table* (IST), there being one such table for each processor in a multiprocessor system. The method of saving program and system status depends on whether the stack has to be switched or not. The IST has entries for each interrupt class, giving the values that must be loaded into the relevant hardware registers for the interrupt processing to proceed. Some details of the various classes of interrupts are given in Appendix 3.

6. Procedure entry

There are three ways of entering a procedure. To enter a procedure running in the same name space and in the same protection environment a simple jump and link instruction may be used; no interrupt is involved. A similar result may be produced without an explicit jump instruction through the operation of the escape mechanism. This occurs when, during the course of the execution of an instruction, an escape descriptor is found to be in, or comes to enter, DR. Execution of the instruction is then cancelled and the computer proceeds to execute the code pointed to by the escape descriptor. It is possible for the code to be written in such a way that eventually re-execution of the cancelled instruction is initiated and execution of the original program is resumed. As an example of the use of this mechanism, suppose that the descriptor pointing to a segment is replaced by an escape descriptor. The effect will be that any attempt to access the segment in question will lead to an escape jump to a routine that might, for example, cause diagnostic information to be printed.

The last act of the routine before returning would be to reinstate the original segment descriptor and set a bit in PSR indicating that the instruction on which the escape jump occurred is to be re-entered.

The CALL instruction is used to make a full dress entry into a procedure using the stack. There is first a precall sequence executed in software; the call sequence itself is implemented in hardware. The CALL instruction is used in four ways. If the address is direct, or if it points to a vector descriptor or a code descriptor, a *normal call* takes place. The contents of PSR remains unchanged, that is there is no change in protection or privilege. If a call is made to a more trustworthy procedure (inward call) or to a less trustworthy procedure (outward call), it is necessary that the kernel of the operating system should be entered so that the values of ACR and PRIV can be changed. The address in the call instruction must therefore point to a system call descriptor. This has parameters which are interpreted by the kernel.

In all cases return is by means of an EXIT instruction. In the case of a normal call this can be via a *code descriptor*; the same is true for an inward call since the original values of ACR and PRIV can be picked up from the stack and reloaded by the mechanism of the EXIT instruction. A check is made that this reloading will not lead to an increase in privilege or protection status; this is necessary since the values of ACR and PRIV on the stack may have been overwritten in error at a level of privilege at which writing into the stack is permitted but not into PSR.

In the case of an outward call return must be via a system call descriptor. It is sometimes necessary for parameters to be passed from a higher ACR level to a lower ACR level. The VALIDATE instruction may be used at the lower level to check their validity. This instruction takes the value of ACR put on the stack at call time and compares it with the value of APF for the new segment. The result is indicated by condition code register (CC).

A call instruction may also be used to implement tasking. This is known as a task call and is via a system call descriptor with parameters that are interpreted appropriately by the kernel. Return is by means of another task call.

7. Slave stores

At the discretion of the designer of the hardware for a particular model in the range, small high speed slave stores may be associated with the main store so as to reduce the average access time; these are often referred to as buffer, or cache stores, and may work either on the associative principle, on the imaging principle, or on a combination of the two. The following are examples of slave stores that may be found in the various models of the 2900 series: instruction slave store; stack slave store; operand slave store. The instruction slave store can contain items only from the segment from which code is being executed and the stack slave can contain items only from the stack segment defined by the SSN register. The operand slave store can contain items from any other segment. In addition the paging system is provided with the usual slave store designed to reduce the number of repeated references that must be made to the segment and page tables.

The slave stores servicing the main memory are operated in terms of virtual addresses rather than real addresses. It is, therefore, necessary for them to be cleared whenever a change occurs in the mapping from virtual addresses to real addresses, that is on a change from one virtual machine to another. Such clearing can be brought about as a side effect of certain instructions and it is unnecessary to go into the details here. An additional problem arises, however, in the case of shared segments, since an item in the main store may be updated under one virtual address while a slave store retains the old value under another virtual address. A solution to this problem would be to mark all shared segments to which writing access is permitted with a bit in the segment table, and to design the hardware so that items from segments so marked never find their way into a slave store. This, however, would be somewhat drastic and would reduce to an undesirable degree the efficiency with which the slave stores would operate.

Instead the items are allowed to pass into the slave store in the ordinary way, carrying with them the indicating bit, and advantage is taken of the fact that the writer of system software will, as a matter of course, protect any section of a shared segment that he needs to update by means of semaphores. It is a rule that he should make use for this purpose of the INCREMENT AND TEST instruction and the TEST AND DECREMENT instruction. These instructions, which are primitives at the hardware level, have the side effect of clearing from the slave stores items marked as coming from shared segments. The mechanism just described is capable of handling the problem that arises when two processors, each with its own operand slave (which may include a write buffer), write into the same area of memory.

A special instruction is provided for the purpose of clearing the address translation slave store of a processor other than that in which the instruction is executed. The specified processor remains halted until it receives a restart signal from the processor which halted it. This facility is available for use by the software when a processor makes a change to a segment table that is also being used by a process running in another processor.

8. Control of peripheral equipment

The principle is adopted in the 2900 range of relying on autonomous, asynchronous, peripheral controllers which are subordinate to the central processor. Input and output activities are always initiated by processes within the central processor. Inward connection (for example, of an online terminal) is achieved by an initial interrupt driven exchange which results in the establishment of a process in the central processor or the connection of the device to an existing process (for example, to a spooling process). A single input/output instruction is provided; the details of the required operations and storage addresses concerned are contained in data tables which are interpreted by the controller.

Since an input/output system cannot afford to wait while virtual to real address translation takes place before each store access, it is performed in the peripheral controller before the transfer begins. In fact the translation only needs to be done once, but clearly the main store area to or from which data is moved must not be paged out. It is therefore locked down in advance of initiation of I/O by the supervisor.

This it does by setting appropriate digits in the page or segment tables concerned. Slave stores whose contents have been invalidated by a peripheral input transfer will normally be cleared by the ACTIVATE instruction which restarts the process waiting to access the store area concerned.

There are four kinds of peripheral controller, each capable of driving a number of device controllers; these are:

General peripheral controller	—electromechanical devices
Disc file controller	—moving head discs
Sectored file controller	—fixed head discs or drums
Communications peripheral controller	—the communications subsystem

Each peripheral controller is assigned a *communication area* in the high speed store. This is established during system initialisation and contains (in fixed format) descriptors pointing to the control blocks which define the required input/output operations for the various device controllers. The communication areas and control blocks are accessible both by the central processor and by the controller concerned, access being controlled by means of a semaphore located in the first word of the area. The communication area also provides space into which termination responses are written by the controller. Synchronisation between the central processor and a controller is achieved by means of interrupts which, in the case of multiprocessor configurations, are broadcast to all central processors. The interrupts may either be taken as they occur or in a group after a preset time interval.

9. Considerations underlying the design

The 2900 series was designed against a background of falling costs of hardware relative to software. The provision of hardware features designed to support software could therefore be contemplated without misgivings. It was felt that the adoption of a stack architecture in the particular form described in this paper would, in addition to providing automatic dynamic allocation of workspace and local name space, facilitate the provision of the following features.

1. Hardware support for a procedure call and parameter passing mechanism.
2. Treatment of procedure calls and interrupts in a uniform manner as forced procedure entries which could be largely handled within the current stack and register environment, thus avoiding the unnecessary suspension and reactivation of processes.

Although in any particular model of the range the exact use that is made of slave stores is at the designer's discretion, slave stores had a definite place in the conception of the architecture. The architecture allows for the provision of local slaves fulfilling special functions in various places and indeed relies upon such slaves being provided for its efficient implementation. For example, it has been common practice in the past to provide the processor with a number of internal registers (accumulators, pointer registers, index registers), primarily to reduce the number of main storage access cycles needed below what would otherwise be necessary and thus increase the operating speed. The 2900 processor, on the other hand, contains a minimum of such registers, since an equivalent reduction in storage access time can be obtained by the provision of efficient slave stores. The reduction in the number of internal processor registers simplifies the task of the writer of compilers, since he is spared the problem of optimising their use.

The use of separate slave stores for separate functions enables any loss of efficiency occasioned by forced clearing of the slaves to be kept to a minimum. For example, the slave associated with the stack need only be cleared when the stack is switched.

Certain instructions in the instruction set have the side effect of bringing about the selective clearing of slave stores.

The protection system follows recently introduced practice of having a number of levels with efficient means provided for switching from one to the other. The hardware supports execute-only access as well as the more usual read-only and read-write access. Consideration was given to a more general protection system of the capability type, but it was felt that the benefits (aside from software and other problems involved in using such a system) were not sufficiently well established to justify its adoption. A conscious attempt has been made to provide a comprehensive set of internal interrupts some of which bring about the switching of the stack and some do not.

On the basis of this it has proved possible to design software that is tolerant of the occurrence of error conditions and other unscheduled events.

Acknowledgements

Whilst making no claim to have put forward any of the features outlined, the authors wish to record that the architecture was developed by many individuals over a period from proposals originally made by (amongst others): J. W. Bowthorpe, J. K. Buckle, J. Connett, (the late) R. D. Feather, D. Howarth, B. J. Moore, V. Pasquali, M. R. Patel, C. B. Taylor and M. R. Wetherfield.

They wish particularly to thank Professor M. V. Wilkes for his help and advice in the preparation of this paper. Finally, they record their thanks to ICL for permission to publish it.

Appendix 1 Registers in the image store

The number of bits in each register is given in brackets.

Visible registers

ACC	Accumulator (32, 64, or 128)
B	Index accumulator (32)
DR	Descriptor register (64)
LNB	Local name base register (16)
CTB	Cross-reference table base register (30)
PC	Program counter (31)
RTC	Real time clock (64)
SF	Stack front pointer register (16)

Invisible registers

IC	Instruction counter (24)	(counts down by 1 for each instruction executed)
IT	Interval timer (24)	(counts down by 1 each n microseconds, where $n < 16$ is given in the hardware manual for the model concerned)
LSTB	Local segment table base register (29)	
PSR	Program status register (24); principal components are:	
	ACR access control register (4)	
	PRIV Privilege (1)	
	OV Overflow (1)	
	PM Program interrupt mask (8)	
	ACS Accumulator size (2)	
PSTB	Public segment table base register (29)	
SSN	Stack segment number register (14)	
SSR	System status register (31); principal components are:	
	II Instruction incomplete indicator (1)	
	RAM Real address node (1)	
	PI Processor identification (2)	
	EP Event pending indicator (1)	
	DGW Diagnostic write (1)	
	ISR Image store read (1)	
	IM System interrupt mask (12)	

Appendix 2 Operand addressing

An instruction consists of an operation code, a literal, and certain tag bits that indicate how the physical address to be presented to the access circuits of the store is evaluated. It will not be necessary to explain how instructions are expressed in binary form, but **Table 1** will show what addresses may appear in them.

DR stands for the content of the second word of the descriptor in DR; otherwise the names of the registers stand for their contents. N stands for the literal in the instruction. Brackets here indicate indirection. For example, ((LNB + N) + B), (TOS + B), and (DR + B) indicate items pointed at by pointers in descriptors held respectively in LNB + N, at the top of the stack, and in DR; '+B' indicates that the pointer concerned is modified by B.

Table 1 Addresses that may appear in instructions

	Access via descriptor		
N(literal)*	—	—	(DR+N)
(LNB+N)*	((LNB+N))*	((LNB+N)+B)	(DR+(LNB+N))
(XNB+N)	((XNB+N))	((XNB+N)+B)	(DR+(XNB+N))
(CTB+N)	((CTB+N))	((CTB+N)+B)	(DR+(CTB+N))
(PC+N)	((PC+N))	((PC+N)+B)	(DR+(PC+N))
TOS	(TOS)	(TOS+B)	(DR+TOS)
B	(DR)	(DR+B)	—

Instructions containing addresses given in the first five lines of the table require 32 bits (of which 18 are allocated to N), except that if N occupies less than eight bits, those marked with a star may be expressed in 16 bits. Instructions with addresses given in the last two rows of the table require 16 bits only. Instructions are also provided for performing operations on strings of consecutive bytes, these strings being addressed by vector or string descriptors held in DR and ACC.

Appendix 3 Interrupt classes

The interrupt classes are tabulated as follows:

Class	*Priority*	*Masking rules*	*Synch/ asynch*	*Stack switched/not switched*
1. System Error	1	—	A/S	SW
2. External	2	2	A	SW
3. Multiprocessor	3	2	A	SW
4. Peripheral	4	2	A	SW
5. Virtual Store		1	S	SW
6. Interval Timer	5	2A	A	SW
7. Program Error		1	S	SW
8. System Call		1	S	N
9. Out		1	S	SW
10. Extracodes		1	S	N
11. Event Pending		3	S	SW
12. Instruction Counter	6	2A	A	N

Priority:		1 is high
Masking Rule:	1.	System error, if masked
	2.	If masked remains pending to system
	2A.	If masked remains pending to process
	3.	If masked ignored.

System error	—Hardware detected errors and violation of Masking rule 1.
External	—Interrupts from devices not having a connection to store.
Multiprocessor	—Interrupts between processors sharing the same store.

Peripheral	—Interrupts from peripheral controller via the store access controller.
Virtual store	—Access requested to a legitimate page or segment which is not in virtual store.
Timer	—Interval timer.
Program error	—Interrupt due to illegal use of instructions or data.
System call	—Use of a system call descriptor in CALL or EXIT instruction.
OUT	—A software generated interrupt.
Extracode	—To allow the execution of selected instructions by software.
Event pending	—
Instruction counter	—Interrupt when IC goes negative.

Book reviews

Introduction to Communication Command and Control Systems, by D. J. Morris, 1977; 350 pages. (*Pergamon Press*, £15·00)

This book brings together the many aspects of communication command and control in an orderly and authoritative manner. In an easily understood and pleasing style the author introduces his subject and gives the reader an overview of system design concepts. We are soon aware that people are an important part of any communication system whether they are the sponsors, the designers or those responsible for the day-to-day running of the network. It is pleasing that this is made clear early in the book and as the author states 'the system should be introduced in stages and not with a sudden revolutionary change', which has relevance to all parties concerned with the implementation.

There are particularly good chapters on sensor base data collection and data transmission theory, the former perhaps better understood as telemetry. However, these are just part of a large subject and the author takes us by stages through the advantages and disadvantages of various multiplexor and concentrator systems until we understand the principle of operation.

We are given an introduction to switching centres and the facilities they can provide and communications network heirarchy which provides much food for thought, but needs to be taken slowly.

There is an interesting chapter on loop transmission, a more recent approach to the design of digital data networks; some of the problems posed and the potential for increasing the data content of transmissions. This is followed by a description of computers in command and control systems and, quite apart from the communications aspects, this is a useful study of computer systems.

Of interest to management (and perhaps the focal point of the book) will be the chapter on distributed computer resources, describing how computer power, peripheral equipment, files and libraries might be shared by many sites in a network; the centralised and distributive techniques and some of the social and political problems to be overcome.

There is a great deal more including the ever-present subjects of secrecy, security and privacy, in fact much to interest a wider readership than the prospective system designers and management personnel to whom this book is directed.

The book is well organised and amply provided with clear diagrams, references and indexes and should prove to be a useful work of reference.

In conclusion it can be observed that we are told not only the benefits of communication command and control, but are frequently warned of the pitfalls along the path to successful implementation, and for this we should be grateful.

R. W. Billett (Dunstable)

Queueing Systems, Volume 2: Computer Applications, by L. Kleinrock, 1976; 549 pages. (*Wiley-Interscience*, £18·00)

The second volume of Professor Kleinrock's text on queueing systems far exceeds the image formed in this reviewer's mind's eye by the spectacle of Volume 1. In essence it presents a range of resource allocation and sharing problems, the meat of operational research, arising from the operation and design of computer systems; problems, moreover, which fall within the context of a queueing theoretic approach for their formulation if not solution. The operational research student and practitioner, for whom the text is eminently suitable, will achieve a double benefit: he will feel on the one hand the challenge of problem areas, in a field with which he is undoubtedly nominally familiar, bristling with thorny unsolved problems; on the other he will obtain an improved understanding (in some cases an initial understanding) of the difficulties of computer system design not least in receiving initiation into the mysteries of that vocabulary and symbolism which makes communication between computer specialists and other mortals sometimes an impossibility. For the computer scientist and systems analyst too the book has much to offer. His problems are expounded in the possibly unfamiliar language of mathematics and he may be put into the position of realising for the first time to whom he should turn for first discussion and consultation in the case of certain difficulties which he may have felt intractable.

This volume has been written in such a way as to stand alone. It is of course expensive enough, but one does not have to buy Volume 1 as well. On the other hand an owner of Volume 1 must feel ownership of Volume 2 to be an almost irresistible allure. Basic results in queueing theory are collected summarily in Chapter 1. Chapter 2, of particular interest independently of computing matters, deals with some approximate methods, in particular with diffusion models and the heavy traffic situation. Chapter 3, concerned with aspects of priority systems, to which the author has made notable original contributions, completes the introduction to the major two thirds of the book. This contains a further three chapters; Chapter 4 deals with time sharing and multiaccess; Chapter 5 deals with communication, network analysis and design; Chapter 6 with measurement, flow and traps. Those concerned with the provision of computing resources, and faced with the mounting costs of communication, will find here not only an absorbing account of technology but material for profound reflection.

In dealing at such a level with computer networks the book is probably unique and derives much from the experience, enthusiasm and involvement of the author. This is enlightening, contagious and irresistible. This reviewer imparts a high commendation.

Brian Conolly (London)