THE HARDWARE OF THE KDF9

by **BILL FINDLAY**

1: KDF9

1.1: BACKGROUND AND OVERVIEW

Announced in 1960, the English Electric KDF9 [Davis60] was one of the most successful products of the early UK computer industry. Even in an era of architectural experimentation, the designers of the KDF9 were bold and innovative. The CPU used separate hardware stacks for expression evaluation and for subroutine linkage. Its three concurrently-running control units shared the work of fetching, decoding, and executing machine code instructions, synchronized by a variety of hardware interlocks. The KDF9 was one of the earliest fully hardware-secured multiprogramming systems. Up to four programs could be run at once under the control of its elegantly simple operating system, the **Timesharing Director**. Each program was confined to its own core area by hardware relocation, and had its own sets of working registers, which were activated when the program was dispatched, so that context switching was efficient. A program could drive I/O devices directly, but was limited by hardware checks to those the Director had allocated to it.

This paper is one of a trilogy. Companion papers provide a synoptic description of the KDF9's architecture and software [Findlay11a]; and a detailed description of the KDF9's various software systems [Findlayxxx]. Here I focus on the hardware, especially on the instruction set and its implementation. It may be useful to read [Findlay11a] before the present work, which deepens that account, but it is not in the least necessary, as all essential material is repeated. My primary sources are EE internal engineering reports [EEC62a, b, c] that are now available on the Internet, and the nearest thing we have to a KDF9 reference manual: the *KDF9 Programming Manual* [EEC69].

1.2: AUTONOMOUS UNITS AND THEIR STORES

A KDF9 computer has three primary and many secondary control units, all microprogrammed and running in parallel with each other, subject to appropriate interlocks. A useful way to think of its large-scale structure is as a community of cooperating finite-state machines. The whole complex works to the beat of a clock giving two 0.25μs pulse trains (P1 and P2) separated by 0.5μs.

The primary control units are **Main Control** (MC), **Arithmetic Control** (AC), and **I/O Control** (IOC).

MC takes most of the responsibility for instruction sequencing, including instruction fetching, jumps, subroutines, and interrupts; for mediating access to the KDF9's various stores; and for address generation and indexing. The **main** (**core**) **store** has up to 32768 words, each of 48 bits. The **Q Store** is a bank of 16 index registers, each of 48 bits. The **Subroutine Jump Nesting Store** (**SJNS**) is a stack of 16-bit return addresses, with maximum depth 17 **links**.

AC directly executes simple ALU instructions, and delegates to the Multiplier/Divider unit and the Shift unit, for more complex arithmetic operations. AC operands are taken from, and results returned to, the **nesting store** (or **nest**), which is an expression-evaluation stack with maximum depth 19 words. In addition there is a 1-bit **Overflow Register**, typically set when the range of a result is exceeded, and a 1-bit Boolean **Test Register**.

Note the absence of a conventional 'program counter' register. This is explained later.

IOC supervises up to 16 DMA channels (known as **buffers** in KDF9 terminology) that are capable of simultaneous, independent transfers. Each buffer operates under the control of its own autonomous microprogram. IOC is responsible for the **Lock-Out Store**—one bit to every 32 words of main store, for mutual exclusion of CPU and I/O transfers.

The Lock-Out Store, Q Store, SJNS and nest are implemented with the same technology: as core storage with a read cycle time of 1μs and a write cycle time of 1.5μs. The main store is comparatively slow, having a read or write cycle time of 6μs, and the architecture of the KDF9 goes to some lengths to mitigate the effect this has on performance.

None of these stores have parity checking.

A hardware **timesharing** option replicates the nest, SJNS and Q Store, so that four register sets are provided, thereby obviating the need to save and restore 48 registers when context-switching between a maximum of four concurrently running processes or **problem programs**. KDF9 runs under the supervision of a Director program, a set of privileged routines normally resident at the bottom of store. Its main function is to respond to interrupts, and thereby provide essential services to problem programs, including management of the timesharing facility. (In the UK computer parlance of the early 1960s, a 'timesharing' system implements multiprogramming, not multi-user interactivity.)

A more unusual type of storage holds the microprogram in the **sequence units**, which direct the actions of MC, AC and IOC. The bits in these stores are fabricated by **pulse transformers**. These are large ferrite toroids, about 1cm in diameter, with multiple secondary windings. When a transformer's primary winding is pulsed, its secondary windings emit a variety of control signals.

The fastest storage elements, registers accessed in critical paths, are made from transistor flip-flops. Transistors are also used in logic gates and as amplifiers for the outputs from core stores and pulse transformers. A KDF9 includes something on the order of 20,000 transistors—a remarkably low figure for such a sophisticated machine, and testimony to its philosophy of gaining speed through clever design instead of heroic expenditure on hardware. Perhaps we glimpse the ghost of Turing, whose computer designs were predicated on exactly the same attitude.

1.3: DATA FORMATS

A KDF9 data word consists of 48 bits numbered from D0 (the most significant bit) to D47 (the least significant). Similar big endian numbering is applied to part-word bit fields. Several data formats are supported:

- as a 48-bit, 2's complement binary integer, fraction, or fixed-point number
- as a 48-bit floating point number, consisting of a sign in D0, an excess-128 exponent in D1-D8, and a mantissa in D9 to D47; where the mantissa, together with D0, forms a 2's complement fraction of 40 bits
- as half of a 96-bit, 2's complement binary integer, fraction, or fixed-point number, where D48 (D0 of the second word) is set to zero, so that the effective capacity is 95 bits
- as half of a 96-bit floating point number, consisting of sign in D0, excess-128 exponent in D1-D8, a first part of the mantissa in D9 to D47, D48 set to zero, and a second part of the mantissa in D57-D95; where D49-D56 are set equal to (exponent – 39) unless that would be negative, in which case the whole of D48-D95 is set to 0; and where the mantissa, together with D0, forms a 2's complement fraction of 79 bits
- as two 24-bit **halfwords**
- as eight characters of 6 bits each
- as three 16-bit 2's complement integers that constitute the **Counter** (**C-part**, D0-D15), the **Increment** (**I-part**, D16-D31) and the **Modifier** (**M-part**, D32-D47) of an indexing word in **Q Store format**
- as a control word for I/O instructions, in which the C-part is a device number; the I-part a main store start-address, or unused; and the M-part a main store end-address, or an operation count, or unused

Single-precision floating-point results are usually rounded: 1 is added to D47 of the result if the first truncated bit is not 0; but double-precision results are always unrounded: the truncated bits are ignored. Floating-point results are always **standardised** (normalised, so that D0 ≠ D9); the only floating point operation that gives a well defined result from a non-standardised operand is the STAND instruction, which effects normalisation.

There are no double-word fetch or store instructions: double words need a pair of fetch or store instructions to copy them into or out of the nest. The less significant word of a pair is held in the deeper of its two nest cells.

There are instructions for fetching and storing 24-bit halfwords. Fetching a halfword expands it to a full word in the nest by filling in the 24 least significant bits with zeros. Storing a halfword from a word in the nest extracts its most significant 24 bits. A halfword can hold a 24-bit 2's complement binary integer, fraction, or fixed-point number, or a floating-point number (the latter consisting of the most significant 24 bits of the 48-bit floating-point format).

I/O devices employ 6-bit characters, up to eight of which can be held in one word; however there are no facilities to address packed characters in main store. Characters are packed into words beginning at the most significant six bits. Conveniently, in all the KDF9 character codes, six zero bits represent a blank (space) character; and six one bits represent a **filler** character, which legible output devices such as the line printer and Flexowriter completely suppress. For a full listing of the character codes used by the various I/O devices, see Appendix 4.

1.4: INSTRUCTION FORMATS

A KDF9 instruction consists of one, two or three **syllables** of eight bits, the first two bits of each instruction giving its type, which is 00_2 for one-syllable instructions, 01_2 for two-syllable instructions, 10_2 for three-syllable jumps, and 11_2 for 'directly addressed' fetch and store instructions of three syllables. Each instruction word is therefore capable of holding between two and six instructions, dependent on their lengths.

One-syllable instructions do not contain an operand address or other parameter, and operate only on the nest in 'Reverse Polish' style. They are carried out by AC (Arithmetic Control). Instructions of two or three syllables are primarily the responsibility of MC (Main Control). Two-syllable operations include all operations that require one or more Q Store numbers—'indirect' memory fetches and stores, operations on the contents of Q Stores, shift orders, I/O orders, and the special J*r*C*p*NZS jump instruction. Three-syllable jumps contain a 16-bit instruction address. Directly addressed fetch and store orders contain a 15-bit word address, and the SET instruction contains a 16-bit constant.

The KDF9 assembly language, called **Usercode** [EEC69], is very unusual in having a 'distributed' syntax that embeds parameters within the Usercode order. For example, the J10C2NZ instruction means '**J**ump to label **10** if the **C**-part of Q Store **2** is **N**ot **Z**ero'. It is possible that this format was suggested by the order code, which distributes opcode and address bits around the machine instruction in an equally unconventional manner—see Appendix 2.

Usercode instructions are labelled by integers. An asterisk preceding a label forces the following, labelled, instruction to start at syllable 0 of a fresh word, any unused syllables of the preceding word being padded with DUMMY (no-op) instructions. This is necessary for the target label of a J*r*C*p*NZS instruction, which does not contain an address, but jumps to syllable 0 of the word before that containing the J*r*C*p*NZS itself.

2: THE CPU

2.1: MAIN CONTROL, ARITHMETIC CONTROL AND I/O CONTROL

KDF9 comprises two blocks: the I/O block and the Computing block. The I/O block is provided with instructions by the Computing block, but otherwise works independently, taking priority for access to main store. The Computing block is composed of MC and AC. MC works collaboratively with AC. They interact most strongly on conditional jumps, where AC computes the Boolean; on shifts, where MC calculates the shift length and AC performs the shift; and on fetches/stores, where MC transfers data to/from the store and AC transfers it to/from the nest. The latter, in particular, does much to compensate for the relatively slow core store.

For a fetch operation, MC initiates a store read. A word is transferred from the store to whichever of two fetch buffers is empty. If both contain data that has not yet been delivered to AC, the transfer is delayed until AC catches up. The fetch buffers mean that MC can complete up to two fetch instructions, their core cycles being overlapped with computation by AC. This is particularly valuable in the very frequently executed inner loops that evaluate scalar products. (KDF9 was very much a 'scientific' computer.) For a store operation, MC saves the destination address in a register; when AC catches up it copies the value from the nest into a store buffer register. When convenient, MC copies the contents of the buffer register to store. Instruction fetching is also the responsibility of MC. There are two instruction-word buffers. MC fetches an instruction word to the instruction-word buffer that is not currently being inspected by AC. Thus instruction-fetching can also overlap with computation by AC.

The art of 'optimum programming' on KDF9 is concerned with the careful relative placement of fetch, store, jump and arithmetic operations, with a view to maximizing the parallelism between store cycles and calculation. In this respect KDF9 is very like modern CPUs.

2.2: Addressing and the Q Store

Problem programs address their instructions and data starting from a virtual effective address of 0. When store is accessed, the hardware offsets this address by the program's starting position in main store. At the same time it checks that the virtual address does not exceed the number of locations allocated to the program by Director. In Director state no offsetting is done, so that Director starts at physical location 0.

The address part of a jump instruction consists of a 13-bit word number and the 3-bit number of a syllable within that word. Consequently, the instructions of a program are confined to its first 8192 words. EXIT (return from subroutine), EXITD (return from Director to a problem program) and OUT (invoke Director from a problem program) all have the same basic format as a jump. EXIT's address part is treated as an offset, to be added to the link in the SJNS.

Effective addresses for data fetches and stores are generated in either of two ways: by 3-syllable instructions containing a 15-bit constant address part and a single Q Store reference, Q*q*; or by 2-syllable instructions containing two Q Store references, Q*k* and Q*q*. The effective address in the first case is the sum of the constant and the contents of M*q*; in the second case it is the sum of M*k* and M*q*. Q0 always contains 0, providing a handy way of getting a zero base.

Data locations in EE Usercode have rigidly stereotyped identifiers. Local variables of subroutines have names of the form V*m*; global variables have names of the form W*m*, Y*m*, YA*m*, …, YZ*m*. If indexed, the identifier is followed by 'M' and the Q Store number; the 2-syllable instructions take the operand form 'M*k*M*q*'. Usercode instructions such as V9; YA7M2; M1M2; and so on, represent data fetches that push values onto the nest; =V9; =YA7M2; =M1M2; and so on, represent data stores that pop values from the nest.

Flags suffixed to the instruction optionally specify index updating, halfword addressing, and 'next word' addressing.

The 'Q' suffix causes the Q*q* register to be updated, *after* the effective address is determined, by adding the contents of I*q* to M*q* and decrementing the contents of C*q*. This allows stepping through a predetermined (but variable) number of locations, at addresses given by an initial address and a predetermined (but variable) stride. There are conditional jump instructions that test whether the C-part of a Q register is (non-)zero, providing for very efficient counting loops.

The 'H' suffix, on a 2-syllable fetch or store order, causes the operand accessed to be a halfword. In this case the content of M*k* is taken as a base word address, and the content of M*q* is taken as a halfword offset, odd-numbered halfwords being those in D24-D47 of the addressed word.

The 'N' suffix, on a 2-syllable fetch or store order, causes the accessed word to be at an address 1 greater than usual (i.e., the next word). This allows efficient processing of arrays of pairs of elements stored sequentially in adjacent words—such as the constituent words of a double-precision number—since Q*q* needs to be updated only once for every word pair, using an increment part set to 2. All combinations of Q, H, and N, in that order, are permitted in a 2-syllable fetch or store order.

The 3-syllable fetch and store orders take 6μs, the 2-syllable fetch and store orders take 7μs, and an additional 1μs is needed for Q register updating in both cases. It is merely coincidental that the time taken by 3-syllable fetch and store orders is the same as the main store read/write cycle time—the core store cycle does not begin until 1.5μs-3.5μs after the start of the instruction and the 6μs it takes is *not* included in the total, as it overlaps with following instructions.

2.3: Expression Evaluation and the Nest

An adder associated with the Q Store carries out the arithmetic involved in Q Store updating and in computing effective addresses. All other calculations, both fixed-point and floating-point, involve the nest. It is common for parameters of a subroutine to be supplied in the nest, especially when the routine implements an arithmetic function.

The top three cells of the nest (N1 at the top, then N2 and N3 as we go deeper) are held in three flip-flop registers. The N registers are managed, transparently to problem programs, by a combination of hardware logic and Director software. When a value is pushed onto the nest, hardware saves N3 in the nest's core stack, N2 in N3, N1 in N2, and the new value in the vacated N1. Popping a value is the simple converse.

At most 16 nest cells are available to a program. The **Nest Over-/Under-flow Violation** (NOUV) interrupt happens after pushing a 17th value, or after popping an empty nest. Since an interrupt is not effected until it is convenient for MC, AC may have performed several further nest operations, leaving the contents of the nest unpredictable. Consequently, no recovery from NOUV is possible for a problem program.

Keeping track of nest depth is one of a KDF9 programmer's main chores, and NOUV interrupts are among the most common results of programming error. In a modular program with deeply nested subroutines it can be difficult to

ensure that NOUV is always avoided. Routines can save (on entry) and restore (before exit), all or part of the nest contents; they are helped in this by conditional jumps that test whether the nest is empty (J*r*EN) or not (J*r*NEN).

It often happens that the order of operands in the nest, while convenient for some purposes, is inconvenient for others. To reorganize the nest, we have the 1-syllable instructions:

- REV: a, b, ... → b, a, ...
- DUP: a, ... → a, a, ...
- ERASE: a, ... → ...
- CAB: a, b, c, ... → c, a, b, ...
- PERM: a, b, c, ... → b, c, a, ...
- REVD: a, b, c, d, ... → c, d, a, b, ...
- DUPD: a, b, ... → a, b, a, b, ...

Simple instructions—including integer +, –, AND, OR, REV, DUP, ERASE, *etc*—take from 1μs to 4μs, 2μs being typical. Floating-point addition takes from 7μs, multiplication takes from 14μs, and division takes from 36μs; the variation being due to data-dependent operand alignment and result normalization. Overflow is set on division by zero and when a numerical result is outside the range of the result type. It remains set until explicitly cleared.

The 1-syllable ZERO and the 3-syllable SET allow for the sourcing of small constants. The most efficient way to push -1 is: ZERO; NOT; and the most efficient way to get +1 is: ZERO; NOT; NEG;—the same number of syllables as SET 1; but slightly faster. Subtracting 1 is best done by: NEG; NOT; and adding 1 by: NOT; NEG. It is nice that these 'hacks' set the overflow in exactly the same cases as SET 1; +; or SET 1; –;.

Some more unusual operations include SIGN, which replaces N1 with –1, 0, or +1, according to its sign; MAX, which arranges that N1≥N2, setting overflow if they were exchanged; and BITS, which replaces N1 with a count of the number of 1-bits it contained (it is interesting to speculate that BITS may stem from Turing's cryptological work). TOB yields a binary integer from a mixed-radix integer and a radix pattern; and FRB does the converse. For a full list of instructions, with execution times, see Appendix 1.

2.4: Control Flow and the SJNS

The J*r* instruction is an unconditional jump to the instruction at label *r*. The instructions: J*r*=Z, J*r*≠Z, J*r*>Z, J*r*≥Z, J*r*<Z, J*r*≤Z test the sign of the top cell of the nest; to compare the top two cells of the nest we have: J*r*= and J*r*≠. All of these pop N1 whether they jump or not; J*r*= and J*r*≠ do not pop N2, being the only dyadic operations with this behaviour. To test whether C*q* is (non-)zero we have: J*r*C*q*Z, J*r*C*q*NZ, and J*r*C*q*NZS. The J*r*V and J*r*NV instructions are conditional on the Overflow Register being (un-)set, while J*r*TR and J*r*NTR are conditional on the Test Register being (un-)set. J*r*(N)V and J*r*(N)TR clear the designated register, whether they jump or not.

J*r*C*q*NZS is known as the **short loop jump**. It jumps, if C*q* is nonzero, to syllable 0 of the instruction word that precedes the word containing the J*r*C*q*NZS instruction; and has the further effect of inhibiting instruction fetch cycles. Thus the loop executes entirely from the instruction word buffers, with no overhead for instruction fetches. Important algorithms, such as scalar product and polynomial evaluation, fit comfortably into the 12 available syllables.

The JS*r* instruction branches unconditionally to the instruction at label *r*, and pushes **its own** address onto the SJNS as a return **link**. This creates issues of SJNS management similar to those that arise with the nest, and the instructions J*r*EJ and J*r*NEJ are provided analogously to J*r*EN and J*r*NEN. The only other instructions that access the SJNS are:

- EXIT *a*: complementary to JS*r*—pops the link from the SJNS, adds the constant *a* (which is an integral multiple of three syllables), and branches to the resultant address
- OUT: pushes its own address onto the SJNS and causes an OUT interrupt, thereby switching control to Director
- EXITD: complementary to OUT, but also used when context-switching—pops the link from the SJNS and branches to the resultant address in program state, with interrupts enabled on completion
- LINK: pops a link from the SJNS and pushes it onto the nest
- =LINK: pops a link from the nest and pushes it onto the SJNS

Normal return from a subroutine is effected by EXIT 1; if there is an abnormal return path, it is taken by EXIT 1; and EXIT 2; is the normal return. Switches are programmed by putting the index into the SJNS, by means of the =LINK instruction, then the EXIT AR*r* instruction jumps to the selected word in the jump table starting at label *r*.

All interrupts, like OUT, use the SJNS for their return address.

3: Input/Output

3.1: Peripheral Devices

Up to 16 I/O buffers can be fitted. Each has its own independent control unit, which is microprogrammed to support its connected device. The monitor console Flexowriter is always on buffer 0, and a paper-tape reader with hardware bootstrap facility is always on buffer 1. Communication with the operators is by means of messages typed on the Flexowriter, which has a button that is pressed to gain the attention of Director by causing a typewriter interrupt. (Merely using the keyboard does not cause an interrupt.)

Peripheral devices fall into two classes: **slow**, or **character** devices; and **fast**, or **word** devices. Buffers for slow devices do a core cycle for each character transferred; fast device buffers do a core cycle for each complete word.

The slow devices include the Flexowriter (10ch/s), paper tape punches (110ch/s), paper tape readers (1000ch/s), card readers (10 card/s), card punches (5 card/s), graph plotters (200 step/s), and line printers (15 line/s).

The native KDF9 paper tape code is rather odd, and makes poor use of the eight punchable locations, or 'channels', in a tape frame. Channels 1-4 and 6-7 encode the six-bit character and channel 5 establishes even parity for the whole frame. But channel 8 is used only to distinguish completely blank tape from the space (SP) character, which has the all-zero code. (Since SP has even parity it would otherwise be represented by a frame with no punching.) Blank tape, and erased tape with all channels punched, are treated as content-less leader or trailer. In normal input modes they are skipped over by the buffer and not transferred. These peculiarities are inherited from the offline Flexowriter tape-punch stations used for data preparation and for the transcription of output tapes to typed copy.

The paper tape code has 'Case Shift' and 'Case Normal' characters to expand the character set. Flexowriters treat them literally—on receiving a Case Shift character, they shift up the type bars so that an alternative glyph can be printed for each code. For example, the data '`Bill Findlay`' would be punched on tape as '`Bß`ILL `ñF`ß`INDLAY`', where I have used **ß** to denote the non-printing Case Shift and **ñ** the non-printing Case Normal. Transcribing a tape containing the characters '`Bß`ILL `ñF`ß`INDLAY`' types out the text '`Bill Findlay`'.

Calcomp drum plotters (models 563, 564, 565 and 566) are an option. They are attached to tape punch buffers, which are manually switchable between punch and plotter. An I/O instruction is provided to test the position of the switch. The plotter takes 6-bit codes that move the pen and/or the paper by one step of fixed size at a time. Strangely, only one diagonal direction is available in a single code; a step on the other diagonal requires separate pen and paper steps. Depending on the particular Calcomp model, steps are of size 0.01 or 0.005 inches, and are taken at the rate of 200 steps/s. Commands to raise or lower the pen take 0.1s. The paper is a roll up to 120 feet in length, and either 11 inches or 29.5 inches wide.

Card readers and punches have two modes of operation. In the 'alphanumeric' mode each column corresponds to one character and is encoded or decoded accordingly. In the 'direct' mode the 12 rows of each column correspond to the 12 bits of two characters and are transferred literally. Line printers use a subset of the alphanumeric card code.

The fast devices are magnetic tape decks, fixed-disc drives, and drum stores.

The magnetic tape physical format uses 16 tracks across the tape: six data tracks, a parity track and a clock track are duplicated for each character. A character is valid if either copy can be read without error. Tapes run at 100inch/s, with 400ch/inch, for a transfer rate of 40kch/s. Inter-block gaps are about 8-9mm long. A faster, but seldom seen, magnetic tape deck runs at double speed (80Kch/s), recording on a steel band instead of plastic tape.

The fixed-disc drive has 16 disc platters, each about 31 inches in diameter. Disc blocks hold 40 words (320 characters); there are 16 blocks per track in an outer recording zone (further from the spindle) and eight per track in an inner zone. The discs spin at 1000 rev/min: the transfer rate in the outer zone is about 85Kch/s, but only half that in the inner zone. Fixed heads are mounted in the outer zone of the first platter, providing access without a seek to a small set of the fastest tracks. Each platter also has its own independent arm, which carries eight read/write heads. Both surfaces of a platter have two heads in the inner track zone and two in the outer zone, so that 96 blocks are available without a seek. Seeks take from 156ms to 367.5ms, with an average of 231ms. The heads can be moved to 64 different positions, giving a capacity of 6144 blocks per platter, or 98304 blocks (31,457,280 characters) per drive. Up to four drives can be attached to the disc buffer, special provision being made for this in its microprogram.

The drum consists of 320 addressable sectors, each of 128 words, for a total of only 40960 words of storage, accessed at a transfer rate of 500kch/s. It is of use mainly for storing frequently executed programs, such as the program source editors and compilers. Few were deployed.

3.2: I/O Instructions

There are two significant aspects to the software control of I/O devices on KDF9: their allocation to problem programs by Director, and their control by problem programs once allocated.

Programs can directly drive I/O devices on buffers allocated to them by Director. Any attempt to access an unallocated buffer causes a **Lock-In Violation** (LIV) interrupt, and termination of the program. This feature of KDF9 means that programs are inherently device-dependent: they must contain logic specific to the type of device they use. This is less of a disadvantage than it might seem, because card reader/punches, paper tape reader/punches, and printers, all use somewhat different characters sets; so programs have to be device-aware for that reason, if no other.

Though program logic is coupled to device type, it is decoupled from device identity. To obtain access to a device, a program asks Director to allocate it one of the type. If such a device is available, Director allocates it to the program and returns its buffer number in N1. The program must save this number for future use.

The instructions that control devices take their parameters from a designated Q Store, Q*q*. C*q* contains the buffer number of the device. (For the fixed disc system, C*q* also contains the seek area number, the platter number and the drive number, taking up all 16 bits.) Depending on the specific instruction, I*q* and M*q* might contain starting and ending *virtual* addresses for a data transfer operation, the transfer count being determined by their difference; or M*q* might contain a repetition count for a control operation; or I*q* and M*q* might both be ignored.

If the least significant bit of an I/O instruction is 1, it sets the device off-line before initiating the operation. The device is only set on-line by the operator pressing a button on its control panel. In this way a program can set up a transfer on a tape reader, for example, which waits until the operator has loaded a tape before attempting to read.

After checking its validity, MC delegates an I/O instruction to its specified buffer, via IOC; if it is a data transfer instruction, MC also provides physical addresses converted from the virtual addresses given in I*q* and M*q*.

There is some attempt at making the effect of a given instruction generic, so that PIAQ*q*, say, acts in a similar way on all input-capable devices, but this is not carried through with complete consistency. Some simpler devices respond to different orders in the same way: PIAQ*q* and PIEQ*q*, for example, have the same effect on a paper tape reader.

A completely generic feature of the I/O architecture is the provision of variable length transfer instructions. These take a (maximum) transfer count, as usual, but terminate early upon transferring an **End Message** character (written '→', code 75_8). When a variable-length read terminates on End Message, any remaining character positions of the last word transferred are set to zero (a SP character). In the case of fast devices, when a variable-length write terminates on End Message, any remaining character positions of the last *block* transferred are set to zero on the output device.

Another feature, generic to the slow devices only, is the availability of the misnamed **character** transfer option. These instructions read one character into, or write one character out of, the least significant bits of each word in the transferred area. This is the only I/O mode that permits the transfer of an arbitrary number of characters without the side effect of termination on End Message. When applied to paper tape, character mode further allows for all eight bits of a frame to be read or written, so that foreign codes can be handled (at the cost of forgoing hardware parity checks).

A peculiarity of the online Flexowriter is that, when a semicolon is written, the write operation changes itself on-the-fly to a read operation; subsequent typed input is transferred to the originally designated output buffer, in character positions following the semicolon. This can be used to program an atomic, prompt-and-response, form of interaction.

The PM*x*Q*q* instructions provide control operations, such as rewinding a magnetic tape or initiating a disc arm seek.

Several instructions enable device-dependent state (e.g., whether a magnetic tape is positioned at the Beginning Of Tape window) to be transferred to the CPU's Test Register; these instructions are also encoded in the PM*x*Q*q* group.

PARQ*q* and BUSYQ*q* both transfer state from a buffer to the Test Register. BUSYQ*q* checks whether a buffer is presently engaged in a transfer. PARQ*q* checks and clears a parity error flag on a buffer's last completed transfer—any other command to a buffer with an uncleared parity error causes a LIV interrupt.

The Usercode programmer is given multiple, device-specific mnemonics for each hardware I/O instruction. For example, the output instruction POAQ*q* can also be written as MWQ*q* when the intended output device is a magnetic tape drive, and as TWQ*q* when it is the console typewriter. The use of a specific mnemonic has no significance at run time; effectively it acts like a comment appended to an unspecific mnemonic in the source program.

Only a fraction of many possible I/O opcodes have defined effects. Moreover, some are defined only for one class of device (input or output). Attempting an undefined I/O operation causes a LIV interrupt. See Appendix 3.

3.3: I/O-DRIVEN MULTIPROGRAMMING

Program blocking by I/O operations is automatically managed by IOC, using a 12-bit **Program Hold-Up** (PHU) register for each of the four program priority levels.

While an I/O transfer is in progress, its core store area is **locked out** by the buffer, so that any attempt to access it concurrently, whether to fetch or store data, to fetch instructions, or use it for another I/O transfer, causes a **Lock-Out Violation** (LOV) interrupt. Attempting an I/O operation on an already-busy buffer also causes a LOV. In response to a LOV, Director suspends the responsible program. One bit of the Lock-Out Store is associated with each **group**, or block of 32 words, so for greatest overlap it is important to ensure that I/O areas are aligned on 32-word boundaries.

The PHU registers are constituted as follows:

- PHU*n*:D11 is set if program level *n* is held up.
- If PHU*n*:D10 is 1 then PHU*n*:D6-D9 contain the number of the buffer on which program level *n* is waiting.
- If PHU*n*:D10 is 0 then there is a core lock-out in effect and PHU*n*:D0-D9 contain the number of the locked-out group.

A buffer records the CPU's privilege state at the start of a transfer. On completion, if Director started the transfer, IOC requests an **End of Director Transfer** (EDT) interrupt; but if the transfer belonged to a problem program, its PHU is cleared. If the cleared PHU belongs to a program of higher priority than the one currently running, IOC requests a **Program** (PR) interrupt. Since a suspended program may be waiting on a shared device (typically, the Flexowriter) made busy by a program of lower priority, every other PHU is then examined to see if it refers to the same buffer. If so, an EDT interrupt is requested instead of PR. This enables Director to take suitable action to resolve the priority inversion over the shared device.

Applying an INTQ*q* instruction to a busy buffer also causes a PR interrupt, effectively yielding the CPU to a program of lower priority; applying INTQ*q* to an idle buffer has no effect. PHU*n*:D0 is set to 1 if an INTQ*q* instruction is responsible for program level *n* being held up.

3.4: I/O CONTROL

IOC provides the interface between the CPU and the I/O devices on their buffers. It contains three sequence units. The ESU1 and ESU2 units provide a conduit for the transfer of data between main store and the buffers. The RSU initializes and finalizes I/O transfers, manages the Lock-Out and PHU Stores, and passes the parameters of the I/O instruction to the relevant buffer. At the end of the transfer the RSU requests the appropriate interrupt from MC.

The **C Store** provides one 48-bit word for each buffer. It contains the transfer's initial, final and current word addresses, the 2-bit priority level at which the transfer was requested, and a flag for Director transfers. The initial address is retained to allow a fast-device operation to be restarted automatically should a parity error be detected during the transfer. Slow device buffers use those 15 bits for different purposes: they hold the next character position within the current word, and indicate transfer options such as 'character' mode.

The **E Store** is a 16×4-bit FIFO that the buffers use to request service from IOC. When it needs attention, a buffer inserts its device number at the end of the E Store. The ESUs take the buffer number at the head of the FIFO, using it to select a C Store and so discover what needs to be done. The IOC can be seen as a multiplexor channel, in more modern language, capable of both byte and word transfers, but not having a 'burst' mode, locked on to a single device. The total I/O rate of the maximum complement of EE devices (8 of the faster tape decks, a disc drive, and a drum store—slow

devices are irrelevant) does not exceed the bandwidth of the core store, so burst mode is not needed. However, one installation [WR67] did modify the E Store mechanism to allocate store cycles on a priority basis, with higher priority for faster devices, their motivation being the need to connect a very fast non-standard device.

4: DIRECTOR STATE

4.1: INTERRUPTS

KDF9 interrupts may be either **voluntary** (OUT or INTQ*q* obeyed by the problem program); **inadvertent** (illegal instruction, nest over-/under-flow, etc); or **housekeeping** (monitor typewriter, clock and I/O device interrupts).

The special register RFIR (Reason For Interrupt Register) records the currently requested interrupt(s). RFIR is inspected from time to time by MC. When an interrupt is accepted, control branches to word 0 of Director, leaving an instruction address in the top cell of the SJNS. On entry to Director state, NIFF (Non-Interrupt Flip-Flop) is set to inhibit further interrupts, but not their recording. However, the RESET interrupt is never inhibited and the NOUV interrupt is completely suppressed. NIFF is also, in effect, the Director-state flag, so that an interruptible Director mode of execution does not exist.

When fetched by the privileged K4 instruction, RFIR is automatically cleared. K4 delivers the following to N1:

- D0-D15: CLOCK COUNT; the integer in D0-D15 is incremented every 32μs; a CLOCK interrupt occurs when overflow from D1 sets D0, but if D0 is already set a RESET interrupt occurs instead
- D22: the PR interrupt is caused by the end of a peripheral transfer started by a program of higher priority than the program currently running; the PR interrupt is also caused by an INTQ*q* instruction applied to a busy device
- D23: the FLEX (Flexowriter) interrupt is caused by the interrupt key on the console typewriter
- D24: the LIV (Lock-In Violation) interrupt is caused by an illegal or privileged instruction, by the use of an unallocated peripheral, by store address wraparound, or by a negative effective address
- D25: the NOUV (Nest Over- or Under-flow Violation) interrupt occurs when an attempt is made to overfill an already-full nest or SJNS, or to pop an already-empty nest or SJNS
- D26: the EDT (End of Director Transfer) interrupt occurs at the end of a peripheral transfer initiated by Director, or if a priority inversion has been detected among programs currently locked-out and waiting for a shared device
- D27: the OUT (system-call) interrupt is caused by the OUT system-call instruction
- D28: the LOV (Lock-Out Violation) interrupt is caused when a program attempts to access any of a locked-out group of 32 words, or when it attempts to command a busy peripheral device other than by INTQ*q*
- D29: the RESET interrupt is caused by jumping to an invalid syllable address (6 or 7), or by a watchdog 'double clock' (implying that a previous CLOCK interrupt has not been handled by Director in the intervening time)

If a store-access instruction with Q Store updating experiences a Lock-Out Violation, the Q register update is suppressed. When the lock-out clears, and the interrupted program is resumed, the instruction can be retried without doubly updating the Q register.

4.2: NESTING-STORE MANAGEMENT BY HARDWARE AND DIRECTOR

The nest depth is represented by a 5-bit counter register. Its least significant four bits provide the address of the cell to be accessed in the nest's 16-word core stack. NOUV is signalled in program state when a push completes with the depth equal to $16+1=21_8$, and when a pop completes with the depth equal to $0-1=37_8$. Since NOUV is completely suppressed when Director is running, it can use all 19 nest cells—the 16 in the core stack and the three flip-flop registers, N1-N3.

Director exploits this to great advantage in its 'short-path' interrupt handler, which is simple enough to be able to work entirely in the N registers, and does not need to save the interrupted program's whole nest to make room for itself. Only the values left in the N registers by the interrupted program need to be saved (in the core stack). This makes context switching in response to PR interrupts rather efficient, the time consumed between interruption and re-entering a program being of the order of 320μs, or about 60 instructions. The 'long path' through Director, where more complex work is done, must save many of the interrupted program's registers; but that happens relatively infrequently.

To maximize hardware speed in a critical path, the destructive reading of a cell in the nest core stack is *not* followed by a write cycle to restore the contents; so reading a cell also clears it. As the nest is a strictly LIFO store, this is not a problem: no attempt can be made to read the cleared value a second time.

Similarly, and again for speed, writing to a cell in the nest core stack is *not* preceded by a read cycle to clear it; so the written bit pattern is effectively OR-ed into the cell. This is valid only if the cell contained zero before the write cycle. Therefore, before entering a program for the first time, Director completely empties its nest—not merely setting the depth to zero, but explicitly clearing the contents. The ERASE order pops the nest, forcing a destructive read from the core stack into N3. By performing enough ERASE orders, Director can ensure that all 19 nest cells are completely zeroized. Subsequent pushes, to a depth less than four, leave the state of the core stack unchanged, although the nest depth increases. The first push that increases the depth to four may cause a non-zero value to be saved in the core stack (at cell 3). This leaves cells 0-2 clear, even when the nest depth reaches 16, the most allowed to a problem program.

Director's short-path interrupt handler saves N1-N3 at zero cost, as a side-effect of pushing three of its own values onto the nest. There is always room for N1-N3 in the core stack—at worst, in cells 0-2. Before returning to an interrupted program, Director performs three ERASE orders, thereby restoring the values that N1-N3 contained before the interrupt. If the interrupted program's nest depth was less than three, one or more non-significant zero words will be fetched into the N registers; since the NOUV interrupt is suppressed in Director, no harm results. The round-trip cost of preserving the nest in the short path is therefore just three ERASE orders, taking 3μs.

If a program causes a NOUV interrupt by overfilling the nest, cells 0-2 of its core stack might be corrupted; but this does not matter, because the program cannot be allowed to continue unless the nest and SJNS are re-initialized.

There is one weakness in the implementation of the nest—the depth is checked only after it changes. A consequence of the reasoning described above is that, for correct LIFO operation, N2 must be clear when the nest depth is 1. But REV, for example, *can* be executed when the depth is only 1, and may swap a non-zero value into N2 without a NOUV. Two pushes later this non-zero value enters the core stack, changing the pristine 0 in cell 2, so that a future write into that cell will be corrupted. However, a program that commits such a gross error is likely to soon attempt a dyadic operation, and will fail at that point. In any case, it affects only itself, and only by making its debugging more difficult.

The implementation of the SJNS is very similar to that of the nest, with a similar division of responsibility between hardware and software. The Jump Buffer, a single flip-flop register analogous to the nest's three N registers, constitutes the 17th cell of the SJNS. It ensures that interrupts can be taken when the SJNS is full, and it promotes the efficiency of the Director's short path. NOUV is caused after a program pushes a 17th link, and after it pops an empty SJNS. The excess link left in the Jump Buffer by a 17th push is overwritten by the NOUV interrupt's return address. Again, this does not matter, because the program cannot be allowed to continue unless the nest and SJNS are re-initialized.

4.3: Director-Only Registers and Instructions

Several special registers, concerned in multiprogramming, are accessible only to Director, although they exercise their influence on the running of problem programs.

The **Base Address** (BA), a 10-bit register, contains the number of the 32-location group of words at which the program starts. It is added to the top ten bits of virtual addresses to effect dynamic relocation. The **Number Of Locations** (NOL), also a 10-bit register, contains the BA-relative group number of the highest addresses the program can validly access. I/O instructions specify virtual addresses in I*q* and M*q*. MC checks that I*q* ≤ M*q*, checks that both are compatible with NOL, and adds to both the value in BA, before passing them on to IOC. A failed check causes a LIV interrupt. On entry to Director BA is set to 0, so Director runs at the bottom of core. Strangely, NOL is still compared with effective addresses in Director state; Director must set NOL to the maximum for the machine, if it wants to guard against causing a LIV interrupt when accessing main store above its own upper limit.

The **Current Peripheral Device Allocation Register** (CPDAR), a 16-bit register, has one bit for each buffer. If a bit is set, the program can validly command the corresponding buffer; if the bit is clear, any such attempt causes a LIV interrupt. The **Current Priority Level** (CPL), a 2-bit register, holds the hardware dispatching priority of the program.

These registers contain values that are proper to one running program, and Director must set them accordingly before (re-)entering that program. The hardware timesharing option provides four instances of the nest, Q Store and SJNS. On context switching between programs, Director can switch quickly between instances by changing the active set number, but must separately save and restore the nest depth, the SJNS depth, BA, NOL, CPL, CPDAR, and the Overflow and Test registers.

There are four of the PHU (Program Hold-Up) registers. For scheduling reasons, Director might want to change the priority level of a program. Since a program's priority fixes the PHU that is used by the buffers to update its dispatching status, Director must wait for all of its I/O transfers to terminate before assigning it to a different priority.

Instructions to manipulate the special registers are available only in Director state. They are:

- EXITD: clear NIFF and jump to the address in the top cell of the SJNS; interrupts are inhibited throughout the execution of EXITD, to allow the machine to adopt a stable state
- CLOQ*q*: clear any lock-outs for the area specified by I*q* and M*q*, and clear PHU*n*, where *n* is the current value of CPL
- TLOQ*q*: test the area specified by I*q* and M*q*, setting the Test Register if any of the encompassed groups are locked out
- CTQ*q*: terminate device activity and clear lock-outs for the transfer specified by Q*q*
- =K0: **if** N1 ≠ 0 **then** switch buzzer on **else** switch buzzer off **end**
- =K1: copy bits N1:D24-D33 to NOL, bits N1:D34-D35 to CPL and bits N1:D38-D47 to BA
- =K2: copy bits N1:D32-D47 to CPDAR, where N1:D32 corresponds to buffer 15 and N1:D47 to buffer 0
- =K3: switch to a new Q Store/nest/SJNS set, with N1:D0-D1 as the new register set number, N1:D2-D6 as the nest depth and N1:D7-D11 as the SJNS depth
- K4: push CLOCK/RFIR onto nest.
- K5: push PHU*i*:D6-D11 onto nest, for *i* in 0..3, with PHU0:D6-D11 in N1:D0-D5, PHU1:D6-D11 in N1:D6-D11, PHU2:D6-D11 in N1:D12-D17, and PHU3:D6-D11 in N1:D18-D23.
- K7: push the current register set number and nest depths, as represented for the =K3 instruction.

The =K3 instruction must be followed by at least six DUMMY instructions, since it needs 6μs to take effect and during this period the machine is in an indeterminate state. All =K*k* instructions *copy* N1, and do not pop it.

Note that register sets are numbered independently of program priority levels. If Director exchanges the priority levels of a pair of programs, it does *not* need to swap the contents of their register sets.

Two other Director-only instructions have to do with I/O: PMGQ*q* and PMHQ*q* do not seem to have more specific Usercode mnemonics, and the machine code for PMGQ*q* is not yet definitely known.

The semantics of PMHQ*q* are clear: it is a 'set lock-out' instruction, the complement of CLOQ*q*. It is used in Director to lock out an area of core as it would have been by a data transfer, but without actually setting up a transfer.

PMGQ*q* does not seem to be used in any extant Director, although it is attested in marginal notes taken during a KDF9 training course, where it is described as 'read C Store'. Perhaps it was used in I/O diagnostic programs.

5: MAIN STORE ORGANISATION

Main store is composed of **modules** of 4K words, each consisting of four 1K **blocks**. A block consists of 48 core planes of 32×32 bits (the Z, X and Y directions, respectively), the whole organised as follows.

Registers associated with the main store include the 15-bit main-store address register (MSAR) and the 48-bit main buffer (MSB). The most significant three bits of MSAR give the module number, the next two give the block within the module, the next five give the X-line and the last five the Y line. A series of decoding matrices interpret each of the fields, giving access either to a further matrix, or, eventually, to the X and Y lines themselves.

The read sequence operates as follows:

1: the matrix system is pulsed to select the X and Y wires. The cores that flip produce a detected signal in the readout wires, which are organised one per XY plane, per module. The output from the read amplifiers is transferred to MSB. If reading is to be inhibited on certain planes (i.e. for a halfword access), an inhibit current cancels the effect of the X-line current and the cores on those planes do not flip.

2: A side effect of the read step is to clear the addressed word, which must therefore be rewritten from MSB. Again the inhibit wires may come into play to safeguard an un-accessed halfword.

The complete read/restore operation is a **core cycle** and takes 6μs.

The write sequence is essentially similar. Step 1 prepares the word by setting it to zero. Step 2 ORs-in its new value, the inhibit wires being used as before to prevent unwanted overwriting during halfword accesses.

6: MICROPROGRAM SEQUENCING

The sequence unit directing a KDF9 subsystem is a finite-state machine. Each state transition emits microprogram control pulses to initiate primitive hardware micro-operations, such as register–to–register transfers. It also emits a new state value that is fed back to the sequence unit. Control and state outputs may be dependent on input signals, allowing for conditional microinstructions and for microprogram looping.

The MC sequence unit, for example, is similar to that used in AC. It has two 8×8 arrays of pulse transformers, and two 6-bit microinstruction registers (MIS1 and MIS2). Activated by a P1 pulse (0.25μs, every 1μs), the contents of MIS1 are decoded to select one of the transformers, which then has its primary winding pulsed: six of its secondary windings respond with outputs that are used to set a successor microinstruction in MIS2. MIS2 similarly puts a new microinstruction into MIS1 on a P2 pulse (like P1, but 0.5μs later). Alternating between MIS1 and MIS2, the sequence unit steps through a microcode procedure that effects the MC contribution to a KDF9 machine code instruction.

As well as their state outputs, the transformers also generate up to two control outputs that trigger micro-operations within MC. Up to four micro-operations can therefore be commanded in each 1μs clock cycle.

Secondary windings can be connected to signals from other parts of the machine, to make their output dependent on conditions external to the sequence unit. For example, we can deduce that at least one transformer taking part in the J*r*<Z microcode has at least one output that depends on N1:D0 (the sign bit of the number at the top of the nest).

The sequence unit technique is very flexible, and offers many opportunities for context-dependent optimization of the transformer arrays. Within an array a given transformer need only be wired with those signals necessary for the effect wanted. A single transformer can used as part of more than one microprogram sequence, with varying effects being conditioned by control signals connected to its secondary windings. Arrays can be configured with just enough transformers for their tasks. This all results in significant economies. For example, the sequence unit for a tape punch buffer contains two arrays of only five transformers each, addressed by 3-bit microinstruction registers.

A basic KDF9, ignoring I/O buffers, uses eight sequence units. MC has one, AC and IOC have three each, and the core store has one to sequence read/write cycles. A fully-specified machine with 16 I/O buffers has no fewer than 24.

7: MAIN CONTROL

The instruction currently being obeyed by MC is held in a 27-bit register called the **instruction staticizer** (INS). INS contains three whole syllables, the first two bits of the following syllable and the first bit of the syllable after that. (AC on the other hand only examines the first syllable of any instruction and so has a 1-syllable staticizer called AINS.) The outputs from INS are used to feed a decoder matrix developing 46 ‘function levels’ which initialize the MIS1 register in the MC sequence unit. That is, they specify entry to different points of the MC sequence unit’s microprogram; these are the ‘S’ numbers listed in Appendixes 1 and 2.

MC is responsible for all two- and three-syllable instructions. Therefore it is responsible for initiating main store cycles, for Q Store operations, for initiating transfers to and from nest, for setting up jumps, for maintaining the SJNS, for initiating I/O operations, and for dealing with interrupts. Instructions that do not involve the nest are implemented entirely by MC, with no participation by AC. Instructions that do involve the nest are executed co-operatively by MC and AC, using buffer and interlock registers to synchronize their actions. These registers are many and various; only the most important are identified in the following.

To implement a write into main store, MC copies the destination address to the Store Buffer Address Register (SBAR); the cycle is initiated after the Store Buffer (SB) is set with a value by AC, at a time when MC next encounters a main store operation.

To implement a read from main store, MC fills MSAR with the source address. If MSAR = SBAR at that point, SB is immediately copied to MSB, thus eliminating a redundant core cycle; otherwise MC initiates a read cycle. When the cycle completes, MC transfers the word from MSB to whichever of the two fetch buffers (FB0, FB1) is free, from whence AC takes the operand for the nest. If both fetch buffers are in use, MC waits for AC to free one of them.

As part of a main store read or write operation, MC also checks for LIV and LOV, in parallel with the actual core cycle if possible. If either condition obtains, the transfer of data is abandoned and the interrupt sequence begins.

To implement an operation on the Q Store, analogous arrangements are made, involving the Q Store Buffer (QB), the Address of item in QB (AQB), and the Q Store Address Register (QSAR). In the case of shift instructions, the shift amount is taken from INS or from C*q* as necessary, and placed in QB for transfer to AC and thence to Shift Control.

A modicum of complexity is introduced into SJNS operations to avoid delays due to updating the SJNS depth counter. The trick is to have two of them, so that the SJNS core stack is addressed alternately by the SJNS Even Address Register (JEVAR) for even-numbered locations, and the SJNS Odd Address Register (JODAR) for odd-numbered locations. During an SJNS operation one counter is used as the current stack pointer while the other is being incremented or decremented to become the new stack pointer. The JTOG flip-flop indicates which of the two stack pointers is to be used. (It is not yet clear whether analogous arrangements are made for the arithmetic nest.)

MC's other great responsibility is instruction fetching. Address registers involved include the Next Instruction-Word Address (NIWA); the Next Instruction (syllable) Address (NIA); and the Current Instruction (syllable) Address (CIA). AC's equivalent to NIA is ANIA and ADIA is its equivalent to CIA. NIWA, NIA, CIA, ANIA and ADIA jointly serve the role of a 'program counter' register in a more conventional architecture.

There are two Instruction-Word Buffers (IWB0 and IWB1), each being a 48-bit flip flop register. MC transfers words containing instructions to whichever of the IWBs is not currently being inspected by AC. CIA is a 4-bit (1+3) register containing the IWB number and the syllable number of the first syllable of the instruction that MC is currently obeying. NIA is like CIA, but holds the address, within the IWBs, of the first syllable of the instruction MC is next to consider. The three extra bits in INS are used by the logic that increments CIA to get the value of NIA. The absolute value of (NIA–CIA) is less than or equal to four, so MC can skip up to four syllables of AC-only instructions within the same IWB, in zero time. If NIA points to the 'other' IWB from CIA, MC is forced to consider the next syllable on from the end of the present instruction, even if it is AC-only, for a 1μs penalty.

An instruction word is transferred to a free IWB as follows:

- If NIA points to the 'other' buffer from CIA, i.e. (NIA:D0 ≠ CIA:D0), and ANIA points to the same buffer as CIA, i.e. (ANAI:D0 = CIA:D0).
- If (NIA:D0 = ANIA:D0) ∧ (NIA:D0 ≠ CIA:D0) then AC is lagging more than six syllables behind, so MC is held up until AC exits its present IWB.
- If (NIA:D1-D3 > 3) ∧ (NIA:D0 = CIA:D0), and the syllable addressed by NIA indicates that the next instruction extends beyond the end of the current IWB, then a new word is fetched into the IWB given by ¬NIA:D0.

Instruction fetching is disabled by a flip-flop called SPIN. If MC finds SPIN set at the start of its cycle, it leaves it set at the end, only unsetting SPIN if it got set during the cycle. This mechanism is used to prevent instruction fetching by the J*r*C*q*NZS short-loop instruction, which keeps SPIN set until C*q* = 0.

The *physical* address of the instruction word to be fetched is held in NIWA, a 13-bit flip-flop register. On unconditional jumps, or successful conditional jumps, the target word's address is converted from virtual to physical form, and the result written to NIWA; its syllable address is copied from INS to NIA. On completion of the instruction fetch cycle, an IWB is filled from MSB.

The jump's target IWB-number and syllable-number are made available to AC in the JAB register. Now, the target syllable number of most jumps is constant, and held in the first syllable of the instruction where it is accessible to AC. EXIT is an exception—its dynamic target address necessitates the JAB register.

Conditional jumps fall into two categories: those that are dependent on values in the nest and those that are not. The latter are implemented entirely in MC. The former require MC to synchronize with AC, which computes the value of the jump condition, passes it to MC, and waits. MC steers instruction fetching accordingly; when it has completely finished with the jump, AC is allowed to continue.

RFIR is inspected from time to time by MC, depending on the instruction being executed. It is not inspected during any of the following, so an interrupt cannot occur in any of those places: an *unsuccessful* conditional jump; shift instructions; Q Store to Q Store transfers; =LINK; OUT; =Q*q*, =RQ*q*, *etc*; and all single-syllable instructions. Interrupts are inhibited throughout the execution of the EXITD instruction, to allow the machine to adopt a stable state.

8: Arithmetic Control

AC obeys 1-syllable instructions and looks at the first syllable of multi-syllable instructions, some of which require AC's participation. The main sequence unit of AC coordinates its activities with MC, and implements simple orders such as nest manipulation, fixed-point addition and subtraction, and logical operations on bit patterns. Shifts are delegated to Shift Control, and multiplication and division to a dedicated Multiplier/Divider. The latter both have their own sequence unit which acts, in effect, as a subroutine of AC.

9: Timing Instruction Sequences

It is difficult to give a definitive execution time for a sequence of KDF9 instructions, because it depends on many dynamically-determined interlock conditions, and these may be influenced by the preceding instruction sequence. The following summarizes the main issues.

- AC does not begin executing operations which MC has not completed.
- AC can execute instructions only after MC has finished with them, but can inspect them earlier.

- AC takes only 1μs to deal with a DUMMY instruction, or one it treats as a DUMMY: these are M±Iq, NCq, DCq, Iq=±1, Iq=±2, Q-to-Q transfers, =Kk, and all I/O instructions.
- AC halts MC when it wants to use the Q Store or the SJNS. More precisely, AC sets NOG and waits for MC to stop, clearing NOG, which cues AC to complete the transfer and restart MC.
- MC waits for AC to compute the condition in a nest-dependent conditional jump; AC then waits until MC has completed the jump.
- MC does not attempt a jump if the JAB interlock is set.
- MC does not initiate a shift if the shift interlock is set.
- MC does not replace the contents of an IWB until AC has finished with it.
- MC does not action store write cycles until SBARM is clear.
- MC does not deal with an instruction of the =Qq class if AC has not dealt with the last such instruction. MC sets AQBM on initiating Q Store housekeeping, and waits at the next such command until AC clears it.
- MC does not action an SJNS instruction until the SJNS interlock is clear; it is set only by =LINK or LINK.
- MC copies previously-fetched data from SB if (SBAR = MSAR) ∧ SBARM.
- MC copies previously-fetched data from QB if (AQB = QSAR) ∧ AQBM.
- MC takes only 1μs to dismiss a 1-syllable instruction it is forced to consider.
- MC waits for AC to catch up before entering an interrupt sequence. Interrupts are requested by setting SKIN, which stops MC at the end of the current instruction. AC stops when it catches up, and the interrupt is effected as soon as SKIN ∧ (NIA = ANIA) ∧ ¬NIFF. This has the curious consequence that an empty-nest NOUV may not be serviced until the nest has been refilled by subsequent AC-only instructions. The link saved on a NOUV interrupt is the address at which NIA equals ANIA, which may be several syllables past the failing instruction.

EXAMPLE 1—TO COMPUTE THE DOUBLE-PRECISION SCALAR PRODUCT $\sum x_i y_i$

On entry to the loop, Q1 = n/1/0, where n is the length of the vectors; the vector $\boldsymbol{x}$ is in the variables YX1…YXn, $\boldsymbol{y}$ is in the variables YY1…YYn; and the nest contains a pair of zeros. The ×+F instruction is a 'multiply and accumulate' operation: it forms the double-precision product N1×N2 and adds that to the double-precision sum in N3 and N4.

```
*1;  YX1M1; YY1M1Q; ×+F;
   J1C1NZS;
```

μs in MC	Instruction	μs in AC
	*1	
6	YX1M1	2
7	YY1M1Q	2
0	×+F	19
4	J1C1NZS	2
17μs	**TOTAL**	**25μs**

The first pass through takes 36μs: AC must wait for MC to finish each of the first two fetches, and MC must set up the short-loop mode when it first encounters the J1C1NZS instruction. Subsequent iterations of the loop take only 25μs: the time in MC is completely overlapped by the time AC takes in the ×+F instruction. Note that a faster core store would not improve the KDF9 time significantly.

The Ferranti Atlas 2, a contemporary of KDF9, allowed some overlap between integer and floating-point units, but otherwise sequenced instructions conventionally. It had much faster floating-point arithmetic than KDF9 and its scalar product loop took from 11.9μs (with 2.5μs core store) to 25.9μs (with 5μs core store).

Using the value of

(*loop time* ÷ *core cycle time*)

as a simple measure of ISP efficiency, KDF9's architecture is between 15% and 20% better in this important algorithm.

EXAMPLE 2—TO COMPUTE THE POLYNOMIAL $\sum a_i x^i$

On entry to the routine N1 contains n, the order of the polynomial; the value x is in N2; and the vector of coefficients $\boldsymbol{a}$ is in the variables YA0…YAn.

```
      DUP; =C1; =M1; I1=-1; =Q2; YA0M1;
   J2C1Z;
      DC1; M+I1;
*1;   Q2; ×F; YA0M1Q; +F;
   J1C1NZS;
2; EXIT 1;
```

Again, the first iteration of the loop takes longer due to once-only setup time. Subsequent iterations of the loop take 28μs, the time in MC being completely overlapped by the floating-point arithmetic. In this case the comparison with Atlas 2 is less favourable to KDF9. Atlas 2 took from 7.4μs to 13.7μs per iteration, its advantage being primarily in the floating-point arithmetic, which is between 12μs and 15μs faster.

APPENDIX 1: KDF9 INSTRUCTION SET AND EXECUTION TIMES

Waits caused by busy-resource interlocks between AC and MC are not included in these times, which are given in μs.

Annotations of the form '(*x*μs)' mean that the entailed core cycle begins *x* μs after MC starts executing the instruction. '*' means that AC must catch up with MC before MC starts executing the order; AC then waits for MC to finish before executing the order itself. '§' means that AC waits for MC to stop, taking at least 2 μs, before it starts executing the order; MC then waits for AC to finish before restarting.

The types of operands are declared using the following indicators: D: 96-bit fixed-point; DF: 96-bit floating-point; F: 48-bit float; FH: 24-bit float; H: 24-bit fixed; I: 48-bit fixed; W: 48 bit non-numerical word. Popping the nest into an operand *x* is denoted ↑*x*; pushing *x* on to the nest is denoted ↓*x*; popping the SJNS is denoted ⇑; pushing the SJNS is denoted ⇓. As an operand, ↑ denotes the popped top of the nest, and ⇑ denotes the popped top of the SJNS. Only net observable effects are shown; no attempt is made to document the actual microcode sequence that implements an order. See [EEC62a] for more complete information on microcode sequences and instruction timing.

	Jumps	AC	MC	Effect
S1	JS*r*	2	11 (4.5μs)	Jump to subroutine: ⇓ NIA
S2	J*r*C*q*Z, J*r*C*q*NZ	2	4 unsuccessful; 11 (4.5μs) successful	Jump if C*q* is (Not) Zero
S3	J*r*	2	8 (1.5μs)	Jump unconditionally
S4	J*r*C*q*NZS	2	4 unsuccessful; 11 (4.5μs) successful first time, but only 4 normally	Jump if C*q* ≠ 0 **S**pecial {to the start of the preceding word in the instruction buffer, and not reloading the latter}
S5*	J*r*=, J*r*≠	2	5 unsuccessful; 12 (5.5μs) successful	Jump if ↑=N2, ↑≠N2, respectively
S6*	J*r*=Z, J*r*≠Z, J*r*<Z, J*r*≥Z, J*r*>Z, J*r*≤Z	2	4 unsuccessful; 11 (4.5μs) successful	Jump if ↑= 0, ↑≠ 0, *etc*;
S7*	J*r*EN, J*r*NEN, J*r*EJ, J*r*NEJ, J*r*TR, J*r*NTR, J*r*V, J*r*NV	2	3 unsuccessful; 10 successful (3.5μs)	Jump if (Not) Empty NEST, (Not) Empty SJNS, (Not) Test Register, (Not) **O**verflow
S8*	EXIT *offset*	2	12 (5.5μs), but 13 if *offset* is an odd number	NIA := (*offset* + ⇑) {return from subroutine / switch to case }
S9*	EXITD	2	12 (5.5μs)	⇑ NIA; NIFF := 0 {Director-only: enter program}
S10	LINK	2	4	↓⇑
S11§	=LINK	2	3	⇓↑

KEY:
r is an instruction label (an integer in Usercode)

	Q Store	AC	MC	Effect
S12	M+I*q*	1	4	M*q* := M*q* + I*q*
S13	M–I*q*	1	5	M*q* := M*q* – I*q*
S14	NC*q*	1	5	C*q* := – C*q*
S15	DC*q*	1	3	C*q* := C*q* – 1
S16	I*q*=1	1	3	I*q* := 1
S17	I*q*=–1	1	3	I*q* := –1
S18	I*q*=2	1	3	I*q* := 2
S19	I*q*=–2	1	3	I*q* := –2
S20	Q*k*TOQ*q*	1	4	Q*q* := Q*k*
S20	C*k*TOQ*q*	1	4	C*q* := C*k*
S20	I*k*TOQ*q*	1	4	I*q* := I*k*
S20	M*k*TOQ*q*	1	4	M*q* := M*k*
S20	IM*k*TOQ*q*	1	4	I*q* := I*k* ; M*q* := M*k*
S20	CM*k*TOQ*q*	1	4	C*q* := C*k* ; M*q* := M*k*
S20	CI*k*TOQ*q*	1	4	C*q* := C*k* ; I*q* := I*p*
S21	Q*q*	2	4	↓ Q*q*
S22	C*q*	2	5	↓ C*q*
S23	I*q*	2	6	↓ I*q*
S24	M*q*	2	4	↓ M*q*

KEY:
k, q are Q Store numbers in the range 0..15

	SET Instruction	AC	MC	Effect
S25	SET*i*	2	4	*i* : I; ↓ *i*

KEY:
i is a constant in the range –32768..+32767

	Q-store	AC	MC	Effect
S26 §	=Q*q*	2	2	↑ Q*q*
S26 §	=C*q*	2	2	↑ C*q*
S26 §	=I*q*	2	2	↑ I*q*
S26 §	=M*q*	2	2	↑ M*q*
S27 §	=RC*q*	2	3	Reset Q*q* to 0/1/0; ↑ C*q*
S27 §	=RI*q*	2	3	Reset Q*q* to 0/1/0; ↑ I*q*
S27 §	=RM*q*	2	3	Reset Q*q* to 0/1/0; ↑ M*q*
S28 §	=+Q*q*	5	5	↓ Q*q*; +; ↑ Q*q*
S29 §	=+C*q*	5	6	↓ C*q*; +; ↑ C*q*
S30 §	=+I*q*	5	7	↓ I*q*; +; ↑ I*q*
S31 §	=+M*q*	5	5	↓ M*q*; +; ↑ M*q*

	Shifts	AC	MC	Effect
S32	SHA±*n*, SHAD±*n*, SHL±*n*, SHLD±*n* where *n* is in the range –64 .. +63	3+*t*	2	Shift top of NEST *n* bits left (+) or right (–), **A**rithmetic (N1), **A**rithmetic **D**ouble-length (N1:N2), **L**ogical (N1), **L**ogical **D**ouble-length (N1:N2)
S32	SHC±*n* where *n* is in the range –48 .. +48	4+*t*	2	Shift N1 *n* bits left (+) or right (–), **C**yclic
S32	×+*n* where *n* is in the range –64 .. +63	16+*t*	2	*r*, *l* : I; *x*, *s* : D; ↑ *r*, *l*; ↑ *s*; ↓ $x = (l \times r) \times 2^n + s$
S33	SHAC*q*, SHADC*q*, SHLC*q*, SHLDC*q* where C*q* is in the range –96 .. +96	3+*t*	3	Shift NEST **C***q* bits left (+) or right (–), **A**rithmetic (N1), **A**rithmetic **D**ouble-length (N1:N2), **L**ogical (N1), **L**ogical **D**ouble-length (N1:N2)
S33	SHCC*q* where C*q* is in the range –48 .. +48	4+*t*	3	Shift N1 **C***q* bits left (+) or right (–), **C**yclic
S33	×+C*q* where C*q* is in the range –96 .. +96	16+*t*	3	*r*, *l* : I; *x*, *s* : D; ↑ *r*, *l*; ↑ *s*; ↓ $x = (l \times r) \times 2^{Cq} + s$

KEY:
t is shifting time in excess of 1*μ*s, given by $t = \lceil a \div 2 \rceil + (\textbf{if } b \neq 0 \textbf{ then } 1 \textbf{ else } 0) - 1$,
such that the absolute value of the shift length $n = 8a + b, 0 \leq a < 12, 0 \leq b < 8$.

	Privileged	AC	MC	Effect
S34 *	=K0	1	3	**if** N1 ≠ 0 **then** switch on buzzer **else** switch off buzzer **end**
	=K1	1	3	Copy N1 to BA, CPL and NOL
	=K2	1	3	Copy N1 to CPDAR
	=K3	1	3	Copy N1 to nest set counters and register set number
S35 *	K4	2	3	↓ RFIR and Clock
	K5	2	3	↓ PHU registers 0-3
	K7	2	3	↓ nest set counters and register set number

The K*k* registers can be accessed only in Director mode, as uncontrolled access may compromise system integrity.

	Fetch and Store	AC	MC	Effect
S36	Direct Fetch: E*a*M*q*, E*a*M*q*Q	2	6 (3.5*μ*s); 7 with Q	↓ [*a*+M*q*] With Q: M*q* := M*q* + I*q*; C*q* := C*q* – 1
S37	Indirect Fetch: M*k*M*q*, M*k*M*q*Q, M*k*M*q*N, M*k*M*q*QN, M*k*M*q*H, M*k*M*q*QH, M*k*M*q*HN, M*k*M*q*QHN	2	7 (4.5*μ*s); 8 with Q	↓ [M*k*+M*q*] With Q: M*q* := M*q* + I*q*; C*q* := C*q* – 1 Halfword M*q* with H, [(1+M*k*)+M*q*] accessed with N.
S38	Direct Store: = E*a*M*q*, = E*a*M*q*Q	1	6 (1.5*μ*s); 7 with Q	↑ [*a*+M*q*] With Q: M*q* := M*q* + I*q*; C*q* := C*q* – 1
S39	Indirect Store: =M*k*M*q*, =M*k*M*q*Q, =M*k*M*q*N, =M*k*M*q*QN, =M*k*M*q*H, =M*k*M*q*QH, =M*k*M*q*HN, =M*k*M*q*QHN	1	7 (1.5*μ*s); 8 with Q	↑ [M*k* +M*q*] With Q: M*q* := M*q* + I*q*; C*q* := C*q* – 1 Halfword M*q* with H, [(1+M*k*)+M*q*] accessed with N.

KEY:
a is a main store address

	I/O	AC	MC
S40	Peripheral Read	1	TLOQ*q*:
S41	Peripheral Write	1	15+*t* PMHQ*q* (?), CLOQ*q*: 16+*t* PI*x*Q*q*, PO*x*Q*q*: 22+*t*, but: 15 if unallocated (LIV) 17 if device busy 18 if uncleared parity error 20 if LOV interrupt
S42	Peripheral Gap	1	MGAPQ*q*/POEQ*q*, MWIPEQ*q*/POFQ*q*: 19, but: 14 if unallocated (LIV), 16 if device busy 17 if uncleared parity error 19 if LOV interrupt
S43	Peripheral Skip	1	MFSKQ*q*/PMAQ*q*, MRWDQ*q*/PMDQ*q*, MBSKQ*q*/PMEQ*q*: as S42
S44	Peripheral Status	1	INTQ*q*: 12, but: 11 if unallocated (LIV) 13 if device busy BUSYQ*q*, CTQ*q*: 13, but: 11 if unallocated (LIV) 13 if device busy PARQ*q*, MBTQ*q*/PMBQ*q*, MLBQ*q*/PMCQ*q*, METQ*q*/PMFQ*q*: 14, but: 11 if unallocated (LIV) 13 if device busy

KEY:
$t = \lceil (Mq - Iq) \div 32 \rceil$ is the time taken setting the lock-out store;
and PI*x*Q*q*, PO*x*Q*q* take an additional 6*μ*s core cycle per character or word, depending on the I/O device type

	I/O	Description
S40	TLOQ*q*	Test Lock-Out
S40	CLOQ*q*	Clear Lock-Out (Director-only)
S40	PIAQ*q*	Read Forward
S40	PIBQ*q*	Read Forward to End-Message
S40	PICQ*q*	Read {Character (Paper Tape, Cards), from Fixed Heads (Disc)}
S40	PIDQ*q*	Read {Character (Paper Tape, Cards), from Fixed Heads (Disc)} to End-Message
S40	PIEQ*q*	Read {Backward (Mag. Tape), Alphanumeric (Cards), Next Sector (Disc)}
S40	PIFQ*q*	Read {Backward (Mag. Tape), Alphanumeric (Cards); Next Sector (Disc)} to End-Message
???	PIGQ*q*	Read {Alphanumeric Character (Cards), Next Sector from Fixed Heads (Disc)}
???	PIHQ*q*	Read {Alphanumeric Character (Cards), Next Sector from Fixed Heads (Disc)} to End-Message
S41	POAQ*q*	Write
S41	POBQ*q*	Write to End-Message
S41	POCQ*q*	Write {Last block (Mag. Tape), Character (Paper Tape, Cards), to Fixed Heads (Disc)}
S41	PODQ*q*	Write {Last block (Mag. Tape), Character (Paper Tape, Cards), to Fixed Heads (Disc)} to End-Message
S42	POEQ*q*	Write gap on output (magnetic- or paper-) tape
S42	POFQ*q*	Wipe long gap on magnetic tape
???	POGQ*q*	Write {Alphanumeric (Cards), Next Sector (Disc)}
???	POHQ*q*	Write {Alphanumeric (Cards), Write Sector (Disc)} to End-Message
???	POKQ*q*	Write {Alphanumeric Character (Cards), Next Sector to Fixed Heads (Disc)} to End-Message
???	POLQ*q*	Write {Alphanumeric Character (Cards), Next Sector to Fixed Heads (Disc)}
S43?	INTQ*q*	Interrupt if device is busy (Director Entry)— may be S44?
S43	PMAQ*q*	Forward Skip (Mag. Tape), Seek (Disc)
S43	PMDQ*q*	Rewind (Mag. Tape), Home Heads (Disc)
S43	PMEQ*q*	Backward Skip (Mag. Tape)
S44	CTQ*q*	Clear Transfer (Director-only)
S44	MANUALQ*q*	Clear Transfer, setting device offline (Director-only)
S44	PARQ*q*	Test for Parity-Check or other device error
S44	BUSYQ*q*	Test for Busy device
S44	PMBQ*q*	Test for Beginning of Tape window (Mag. Tape)
S44	PMCQ*q*	Test for Last Block
S44	PMFQ*q*	Test for End of Tape Warning (Mag. Tape), Test for End of Area (Disc)
???	PMGQ*q*	Read C-Store (Director-only)
???	PMHQ*q*	Set Lock-Out (Director-only)
???	PMKQ*q*	Forward Skip, even parity, on IBM tape
???	PMLQ*q*	Backward Skip, even parity, on IBM tape

	Interrupts	AC	MC	Effect
S45	OUT	2	3 (2.5μs)	System-call: ⇓ NIA; enter Director state
	Interrupt	2	3 (2.5μs)	

'arithmetic', all S46	AC	Effect {MC takes 0μs, or 1μs if too far ahead of AC}
+, –	1	r, l : I; ↑ r, l; ↓ $l \pm r$
ABS	1	r: I; ↑ r; ↓ $\lvert r \rvert$
DUMMY	1	'No-op'
ERASE, REV	1	Reorder top of nest: see §2.3.
NEG	1	r: I; ↑ r; ↓ $-r$
NOT	1	r: W; ↑ r; ↓ $\neg r$
AND, OR	1	r, l : W; ↑ r, l; ↓ $l \wedge r$ {but for OR: ↓ $l \vee r$}
ROUND	1	r: D; ↑ r; ↓ (r rounded to I)
VR	1	Clear Overflow
=TR	2	↑ l; **if** $l < 0$ **then** set Test Register **end**
CONT	2	r: D; ↑ r; ↓ (r contracted to I)
DUP, CAB, PERM, ZERO	2	Reorder top of nest: see §2.3.
NEGD	2	r: D; ↑ r; ↓ $-r$
NEV	2	r, l : W; ↑ r, l; ↓ $l \not\equiv r$ {**N**ot **E**qui**v**alent, i.e., exclusive or}
ROUNDH	2	r: I; ↑ r; ↓ (r rounded to H) {D24-D47 of result are undefined}
+D, –D	3	r, l : D; ↑ r, l; ↓ $l \pm r$
NEGF	3	r: F; ↑ r; ↓ $-r$
ROUNDF	3	r: DF; ↑ r; ↓ (r rounded to F)
ROUNDHF	3	r: F; ↑ r; ↓ (r rounded to FH) {D24-D47 of result are undefined}
SIGN	3	r, l : I; ↑ r, l; ↓ **if** $l > r$ **then** +1 **elsif** $l < r$ **then** –1 **else** 0 **end**
STR	3	r: I; ↑ r; ↓(r stretched to D): D0-D47 of result = r:D0, D48 = 0
ABSF	4	r: F; ↑ r; ↓ $\lvert r \rvert$ {ABSF takes only 1μs if $r \geq 0$}
DUPD, REVD	4	Reorder top of nest: see §2.3.
MAX	4	r, l : I; ↑ r, l; **if** $l \geq r$ **then** ↓ r, l; set Overflow **else** ↓ l, r **end**
SIGNF	5	r, l : F; ↑ r, l; ↓ **if** $l > r$ **then** +1 **elsif** $l < r$ **then** –1 **else** 0 **end**
STAND	5+n	r: F; ↑ r; ↓ (r normalised)
FIX	6	r : F; y, x : I; ↑ r; ↓ y, x: $r = y \times 2^x, -1 \leq y < +1$
MAXF	6	r, l : F; ↑ r, l; **if** $l \geq r$ **then** ↓ r, l; set Overflow **else** ↓ l, r **end**
FLOAT	7+n	x : F; r, l : I; ↑ r, l; ↓ $x = l \times 2^r$: $-128 \leq r \leq +127$
+F, –F	7+a+n	r, l : F; ↑ r, l; ↓ $l \pm r$
FLOATD	8+n	x : DF; r : I; l : D; ↑ r, l; ↓ $x = l \times 2^r$, unrounded: $-128 \leq r \leq +127$
NEGDF	9+n	r: DF; ↑ r; ↓ $-r$, unrounded
+DF, –DF	12+a+n Δ	r, l : DF; ↑ r, l; ↓ $l \pm r$, unrounded
×D	14	r, l : I; x : D; ↑ r, l; ↓ $x = l \times r$
×	15	r, l : I; x : D; ↑ r, l; ↓ D0-D47 of ($x = l \times r$), rounded
×F	15+n	r, l, x : F; ↑ r, l; ↓ $x = l \times r$
×DF	16+n	r, l : F; x : DF; ↑ r, l; ↓ $x = l \times r$, unrounded
×+F	18+a+n	r, l : F; x, s : DF; ↑ r, l; ↑ s; ↓ $x = l \times r + s$, unrounded
BITS	27	r : W; x : I; ↑ r; ↓ x = # of 1-bits in r
÷F	35+n	r, l, x : F; ↑ r, l; ↓ $x = l \div r$, unrounded
÷DF	36+n	x, r : F; l : DF; ↑ r, l; ↓ $x = l \div r$, unrounded
÷	36+a+n	x, r, l : I; ↑ r, l; ↓ $x = l \div r$, rounded: $-1 \leq x < +1$
÷D	36+a+n	x, r : I; l : D; ↑ r, l; ↓ $x = l \div r$, rounded: $-1 \leq x < +1$
÷I	36+a+n	d, n, q, r : I; ↑ d; ↑ n; ↓ q, r: $n = d \times q + r$, $\lvert r \rvert < \lvert d \rvert$, $rd > 0$
÷R	36+a+n	f, x, r : I; l : D; ↑ r, l; ↓ f, x: $l = r \times x + 2^{-47} f$, $-1 \leq x < +1$
TOB	≃ 2+4b	r, l : W; b: I; ↑ r, l; ↓ $b = r$ converted to binary using radixes l
FRB	≃ 8+3b	r : I; l, c : W; ↑ r, l; ↓ $c = r$ converted from binary using radixes l

KEY:

a is shifting time in excess of 1μs needed to **a**lign the operands

n is shifting time in excess of 1μs needed to produce a **n**ormalized result

b is the number of 1 **b**its in the numerical operand

Δ denotes a "variable time" instruction, whose timing depends on which operand has the larger exponent

APPENDIX 2: KDF9 INSTRUCTION SET ENCODING

Operand coding symbols:

a	A main store word address bit
b	A non-decoded "don't care" bit—should be set to 0
h	A halfword is to be added to the return address if h = 0
i	A bit of a 16-bit constant
kkkk, qqqq	Q Store register numbers (0..15)
sss	A instruction syllable number in 0..5; 6 and 7 invalid
u	Device is set unready on completion of operation

The first column gives the identifier of the MC microcode sequence that carries out the instruction.

	Instruction \ Bits:	0, 1	2, 3	4	5–7	8–11	12–15	16–23
S1	JSr	10	00	a	sss	1101	aaaa	aaaaaaaa
S2	J*r*C*q*Z	10	10	a	sss	qqqq	aaaa	aaaaaaaa
S2	J*r*C*q*NZ	10	11	a	sss	qqqq	aaaa	aaaaaaaa
S3	Jr	10	00	a	sss	1011	aaaa	aaaaaaaa
S4	J*r*C*q*NZS	01	11	1	111	qqqq	bbbb	
S5	J*r*=	10	01	a	sss	0001	aaaa	aaaaaaaa
S5	J*r*≠	10	00	a	sss	0001	aaaa	aaaaaaaa
S6	J*r*>Z	10	01	a	sss	0100	aaaa	aaaaaaaa
S6	J*r*≤Z	10	00	a	sss	0100	aaaa	aaaaaaaa
S6	J*r*<Z	10	01	a	sss	0010	aaaa	aaaaaaaa
S6	J*r*≥Z	10	00	a	sss	0010	aaaa	aaaaaaaa
S6	J*r*=Z	10	01	a	sss	0110	aaaa	aaaaaaaa
S6	J*r*≠Z	10	00	a	sss	0110	aaaa	aaaaaaaa
S7	J*r*V	10	01	a	sss	1000	aaaa	aaaaaaaa
S7	J*r*NV	10	00	a	sss	1000	aaaa	aaaaaaaa
S7	J*r*EN	10	01	a	sss	1010	aaaa	aaaaaaaa
S7	J*r*NEN	10	00	a	sss	1010	aaaa	aaaaaaaa
S7	J*r*EJ	10	01	a	sss	1100	aaaa	aaaaaaaa
S7	J*r*NEJ	10	00	a	sss	1100	aaaa	aaaaaaaa
S7	J*r*TR	10	01	a	sss	1110	aaaa	aaaaaaaa
S7	J*r*NTR	10	00	a	sss	1110	aaaa	aaaaaaaa
S8	EXIT	10	00	a	0h0	1111	aaaa	aaaaaaaa
S9	EXITD	10	01	0	010	1111	0000	00000000

S10	LINK	01	11	1	011	0000	bbbb
S11	=LINK	01	11	1	100	0000	bbbb

S12	M+I*q*	01	10	0	000	qqqq	bbbb
S13	M–I*q*	01	10	0	001	qqqq	bbbb
S14	NC*q*	01	10	0	010	qqqq	bbbb
S15	DC*q*	01	10	0	011	qqqq	bbbb
S16	I*q*=1	01	10	0	100	qqqq	bbbb
S17	I*q*=–1	01	10	0	101	qqqq	bbbb
S18	I*q*=2	01	10	0	110	qqqq	bbbb
S19	I*q*=–2	01	10	0	111	qqqq	bbbb
S20	M*k*TOQ*q*	01	10	1	001	kkkk	qqqq
S20	I*k*TOQ*q*	01	10	1	010	kkkk	qqqq
S20	IM*k*TOQ*q*	01	10	1	011	kkkk	qqqq
S20	C*k*TOQ*q*	01	10	1	100	kkkk	qqqq
S20	CM*k*TOQ*q*	01	10	1	101	kkkk	qqqq
S20	CI*k*TOQ*q*	01	10	1	110	kkkk	qqqq
S20	Q*k*TOQ*q*	01	10	1	111	kkkk	qqqq

S21	Q*q*	01	11	1	001	qqqq	111b
S22	C*q*	01	11	1	001	qqqq	100b
S23	I*q*	01	11	1	001	qqqq	010b
S24	M*q*	01	11	1	001	qqqq	001b

S25	SET	11	bb	b	1b0	iiii	iiii	iiiiiiii

S26	=Q*q*	01	11	1	000	qqqq	1110
S26	=C*q*	01	11	1	000	qqqq	1000
S26	=I*q*	01	11	1	000	qqqq	0100
S26	=M*q*	01	11	1	000	qqqq	0010
S27	=RC*q*	01	11	1	000	qqqq	1001
S27	=RI*q*	01	11	1	000	qqqq	0101
S27	=RM*q*	01	11	1	000	qqqq	0011
S28	=+Q*q*	01	11	1	010	qqqq	111b
S29	=+C*q*	01	11	1	010	qqqq	100b
S30	=+I*q*	01	11	1	010	qqqq	010b
S31	=+M*q*	01	11	1	010	qqqq	001b

S32	SHA±*n*	01	11	0	001	nnnn	nnn1
S32	SHAD±*n*	01	11	0	010	nnnn	nnn1
S32	×+±*n*	01	11	0	011	nnnn	nnn1
S32	SHL±*n*	01	11	0	100	nnnn	nnn1
S32	SHLD±*n*	01	11	0	110	nnnn	nnn1
S32	SHC±*n*	01	11	0	111	nnnn	nnn1
S33	SHAC*q*	01	11	0	001	qqqq	bbb0
S33	SHADC*q*	01	11	0	010	qqqq	bbb0
S33	×+C*q*	01	11	0	011	qqqq	bbb0
S33	SHLC*q*	01	11	0	100	qqqq	bbb0
S33	SHLDC*q*	01	11	0	110	qqqq	bbb0
S33	SHCC*q*	01	11	0	111	qqqq	bbb0

S34	=K0	01	11	1	101	1000	0000
S34	=K1	01	11	1	101	0100	0000
S34	=K2	01	11	1	101	0010	0000
S34	=K3	01	11	1	101	0001	0000
S35	K4	01	11	1	110	0000	1000
S35	K5	01	11	1	110	0000	0100
S35	K7	01	11	1	110	0000	0001

S36	E*a*M*q*	11	aa	a	000	qqqq	aaaa	aaaaaaaa
S36	E*a*M*q*Q	11	aa	a	010	qqqq	aaaa	aaaaaaaa

S37	M*k*M*q*	01	00	0	000	qqqq	kkkk
S37	M*k*M*q*Q	01	00	0	010	qqqq	kkkk
S37	M*k*M*q*H	01	00	0	100	qqqq	kkkk
S37	M*k*M*q*QH	01	00	0	110	qqqq	kkkk
S37	M*k*M*q*N	01	00	1	000	qqqq	kkkk
S37	M*k*M*q*QN	01	00	1	010	qqqq	kkkk
S37	M*k*M*q*HN	01	00	1	100	qqqq	kkkk
S37	M*k*M*q*QHN	01	00	1	110	qqqq	kkkk

S38	=E*a*M*q*	11	aa	a	001	qqqq	aaaa	aaaaaaaa
S38	=E*a*M*q*Q	11	aa	a	011	qqqq	aaaa	aaaaaaaa

S39	=M*k*M*q*	01	00	0	001	qqqq	kkkk
S39	=M*k*M*q*Q	01	00	0	011	qqqq	kkkk
S39	=M*k*M*q*H	01	00	0	101	qqqq	kkkk
S39	=M*k*M*q*QH	01	00	0	111	qqqq	kkkk
S39	=M*k*M*q*N	01	00	1	001	qqqq	kkkk
S39	=M*k*M*q*QN	01	00	1	011	qqqq	kkkk
S39	=M*k*M*q*HN	01	00	1	101	qqqq	kkkk
S39	=M*k*M*q*QHN	01	00	1	111	qqqq	kkkk

S40		CLOQ*q*	01	01	0	100	qqqq	001u
S40		TLOQ*q*	01	01	0	100	qqqq	010u
S40	PIA	MFRQ*q*	01	01	0	100	qqqq	000u
S40	PIB	MFREQ*q*	01	01	0	101	qqqq	000u
S40	PIC	RCQ*q*	01	01	0	100	qqqq	100u
S40	PID	RCEQ*q*	01	01	0	101	qqqq	100u
S40	PIE	MBRQ*q*	01	01	0	110	qqqq	000u
S40	PIF	MBREQ*q*	01	01	0	111	qqqq	000u
???	PIG	PIGQ*q*	01	01	0	110	qqqq	100u
???	PIH	PIHQ*q*	01	01	0	111	qqqq	100u

S41	POA	MWQ*q*	01	01	1	000	qqqq	000u
S41	POB	MWEQ*q*	01	01	1	001	qqqq	000u
S41	POC	MLWQ*q*	01	01	1	000	qqqq	100u
S41	POD	MLWEQ*q*	01	01	1	001	qqqq	100u
S42	POE	MGAPQ*q*	01	01	1	000	qqqq	110u
S42	POF	MWIPEQ*q*	01	01	1	000	qqqq	010u
???	POG	POGQ*q*	01	01	1	010	qqqq	000u
???	POH	POHQ*q*	01	01	1	011	qqqq	000u
???	POK	POKQ*q*	01	01	1	011	qqqq	100u
???	POL	POLQ*q*	01	01	1	010	qqqq	100u

S43		INTQ*q*	01	01	1	10b	qqqq	0b1u
S44		CTQ*q*	01	01	0	000	qqqq	0000
S44		MANUALQ*q*	01	01	0	000	qqqq	0001
S44		BUSYQ*q*	01	01	0	000	qqqq	001u
S44		PARQ*q*	01	01	0	001	qqqq	000u
S43	PMA	MFSKQ*q*	01	01	1	100	qqqq	000u
S44	PMB	MBTQ*q*	01	01	0	000	qqqq	100u
S44	PMC	MLBQ*q*	01	01	0	000	qqqq	010u
S43	PMD	MRWDQ*q*	01	01	1	110	qqqq	100u
S43	PME	MBSKQ*q*	01	01	1	110	qqqq	000u
S44	PMF	METQ*q*	01	01	0	010	qqqq	000u
???	PMG	fetch C store	??	??	?	???	qqqq	????
???	PMH	set lock-out	??	??	?	???	qqqq	????
???	PMK	7T FSK *even*	01	01	1	100	qqqq	010u
???	PML	7T BSK *even*	01	01	1	110	qqqq	010u

Note: if PMH, 'set lock-out' is like CLOQ*q* and TLOQ*q*, it *might* have the code `01 01 0 100 qqqq 100u`, or 124:8. PMG remains a complete mystery.

S45	OUT	10	00	b	bbb	1001	bbbb	bbbbbbbb

S46	Illegal	00	000000
S46	VR	00	000001
S46	=TR	00	000010
S46	BITS	00	000011
S46	×F	00	000100
S46	×DF	00	000101
S46	Unused ?	00	000110
S46	×+F	00	000111
S46	NEGD	00	001000
S46	OR	00	001001
S46	PERM	00	001010
S46	TOB	00	001011
S46	ROUNDH	00	001100
S46	NEV	00	001101
S46	ROUND	00	001110
S46	DUMMY	00	001111
S46	ROUNDF	00	010000
S46	ROUNDHF	00	010001
S46	–DF	00	010010
S46	+DF	00	010011
S46	FLOAT	00	010100
S46	FLOATD	00	010101
S46	ABS	00	010110
S46	NEG	00	010111
S46	ABSF	00	011000
S46	NEGF	00	011001
S46	MAX	00	011010
S46	NOT	00	011011
S46	×D	00	011100
S46	×	00	011101
S46	–	00	011110
S46	SIGN	00	011111

S46	Illegal	00	100000
S46	ZERO	00	100001
S46	DUP	00	100010
S46	DUPD	00	100011
S46	÷I	00	100100
S46	FIX	00	100101
S46	Illegal	00	100110
S46	STR	00	100111
S46	CONT	00	101000
S46	REVD	00	101001
S46	ERASE	00	101010
S46	–D	00	101011
S46	AND	00	101100
S46	Illegal	00	101101
S46	+	00	101110
S46	+D	00	101111
S46	÷	00	110000
S46	÷D	00	110001
S46	÷F	00	110010
S46	÷DF	00	110011
S46	÷R	00	110100
S46	REV	00	110101
S46	CAB	00	110110
S46	FRB	00	110111
S46	STAND	00	111000
S46	NEGDF	00	111001
S46	MAXF	00	111010
S46	Illegal	00	111011
S46	+F	00	111100
S46	–F	00	111101
S46	Illegal	00	111110
S46	SIGNF	00	111111

APPENDIX 3: THE I/O INSTRUCTIONS

In the following tables, blank cells are to be taken as inheriting the *common* semantics; where something is shown in a cell it overrides the *common* semantics.

INPUT INSTRUCTIONS

	PIA	PIB	PIC	PID	PIE	PIF	PIG	PIH
common	R	RE	CR	CRE	=PIA	=PIB	=PIC	=PID
CR	BR	BRE	CBR	CBRE	AR	ARE	CAR	CARE
FD			FR	FRE	RN	RNE	FRN	FRNE
MT EE			=PIA	=PIB	<R	<RE	=PIE	=PIF
MT IBM	OR	VR	=PIA	=PIB	O<R	V<R	=PIE	=PIF
SI					XR	XRE		

OUTPUT INSTRUCTIONS

	POA	POB	POC	POD	POE	POF	POG	POH	POK	POL
common	W	WE	CW	CWE	LIV	LIV	UND	UND	UND	UND
CP	BW	BWE	CBW	CBWE	=POC	=POA	AW	AWE	CAWE	CAW
DR			=POA	=POB	WZ	=POE				
FD			FW	FWE	=POC	=POA	WN	WNE	FWNE	FWE
GP		=POA		=POC	NOP	NOP				
LP					???	???	???	???	???	???
MT EE			WL	WLE	GAP	WIPE				
MT IBM	OW	VW	OT	VT	GAP	WIPE				
SI					CGAP	WGAP				
TP					CGAP	WGAP				

DEVICE CONTROL INSTRUCTIONS

	PMA	PMB	PMC	PMD	PME	PMF	PMG	PMH	PMK	PML
common	LIV	NOP	NOP	LIV	LIV	NOP	RCS?	SLO?	UND	UND
CR		TRC?								
FD	S			P	UND	EOA?				
GP/TP		GPA?								
MT EE	>>	BOT?	LBL?	<<<	<<	ETW?				
MT IBM	O>>	BOT?	LBL?	<<<	O<<	ETW?			V>>	V<<
SI		T8T?	T8T?							
TR		T8T?								

KEY:

common	these effects are common to all devices, unless over-ridden by a specific table entry
CP	card punch
CR	card reader
DR	drum store
FD	fixed disc store
GP	graph plotter
LP	line printer
MT EE	EE magnetic tape deck
MT IBM	IBM-compatible 7-track magnetic tape deck
SI	standard interface buffer
TP	paper tape punch
TR	paper tape reader

`A`	alphanumeric (converted to/from all 12 card rows in each column)
`B`	binary (direct to/from 6 card rows, twice in each column)
`C`	character (each character transferred is stored in the least significant 6 or 8 bits of a word)
`E`	stopping on an End Message character
`F`	using the fixed heads
`N`	next sector
`O`	in odd parity
`P`	clear disc head positions
`R`	read
`S`	seek disc heads
`T`	write tape mark
`V`	in even parity
`X`	without parity
`<`	backward (for read operation)
`>>`	forward skip up to M*q* blocks
`<<`	backward skip up to M*q* blocks
`<<<`	rewind the tape to before its first block

`=P`*xy*	having the same effect as the P*xy* order for this device

`CGAP`	punch a gap of length M*q* / 10 inches
`WGAP`	punch a gap? (‘word gap’; undocumented)
`GAP`	erase an inter-block gap of M*q* words
`WIPE`	erase a bigger inter-block gap than `GAP`

`BOT?`	Test Register := tape at Beginning Of Tape window
`EOA?`	Test Register := disc transfer reached end of area
`ETW?`	Test Register := tape at End Tape Warning
`GPA?`	Test Register := a graph plotter is attached to the buffer and not a tape punch
`LBL?`	Test Register := tape at Last Block marker
`T8T?`	Test Register := device in 8-channel mode
`TRC?`	Test Register := recheck switch is set on the card reader

`RCS?`	Director-only (LIV caused in program mode): ‘read C store’
`SLO?`	Director-only (LIV caused in program mode): ‘set lock-outs’
`LIV`	Lock-In Violation (always)
`NOP`	no operation
`UND`	undefined
`???`	undocumented

APPENDIX 4: THE KDF9 CHARACTER SETS AND CODES

Line printer	SP	NLC	CRLF	PC	NLC	NLC	%	'
Normal Case	SP	NLC	CRLF	PC	HT	NLC	CS	CN
Shift Case	SP	NLC	CRLF	PC	HT	NLC	CS	CN
Card rows	*none*	Y68	Y28	028	058	Y58	048	Y78
Octal code	00	01	02	03	04	05	06	07

Line printer	:	=	(	)	£	*	,	/
Normal Case	NLC	NLC	NLC	NLC	NLC	NLC	NLC	/
Shift Case	NLC	NLC	NLC	NLC	NLC	NLC	NLC	:
Card rows	48	38	X58	X28	X38	X48	038	01
Octal code	10	11	12	13	14	15	16	17

Line printer	0	1	2	3	4	5	6	7
Normal Case	0	1	2	3	4	5	6	7
Shift Case	↑	[	]	<	>	=	×	÷
Card rows	0	1	2	3	4	5	6	7
Octal code	20	21	22	23	24	25	26	27

Line printer	8	9	NLC	$_{10}$	;	+	−	.
Normal Case	8	9	_	$_{10}$	;	+	−	.
Shift Case	(	)	_	£	;	≠	*	,
Card rows	8	9	068	X68	Y48	Y	X	Y38
Octal code	30	31	32	33	34	35	36	37

Line printer	NLC	A	B	C	D	E	F	G
Normal Case	NLC	A	B	C	D	E	F	G
Shift Case	NLC	a	b	c	d	e	f	g
Card rows	078	Y1	Y2	Y3	Y4	Y5	Y6	Y7
Octal code	40	41	42	43	44	45	46	47

Line printer	H	I	J	K	L	M	N	O
Normal Case	H	I	J	K	L	M	N	O
Shift Case	h	i	j	k	l	m	n	o
Card rows	Y8	Y9	X1	X2	X3	X4	X5	X6
Octal code	50	51	52	53	54	55	56	57

Line printer	P	Q	R	S	T	U	V	W
Normal Case	P	Q	R	S	T	U	V	W
Shift Case	p	q	r	s	t	u	v	w
Card rows	X7	X8	X9	02	03	04	05	06
Octal code	60	61	62	63	64	65	66	67

Line printer	X	Y	Z	NLC	NLC	NLC	NLC	Ø
Normal Case	X	Y	Z	NLC	NLC	→	NLC	Ø
Shift Case	x	y	z	NLC	NLC	→	NLC	Ø
Card rows	07	08	09	58	68	78	X78	28
Octal code	70	71	72	73	74	75	76	77

KEY:
SP is (blank) Space; CRLF is Carriage Return Line Feed (i.e. New Line); PC is Page Change (i.e. Form Feed); HT is Horizontal Tab; CS is Case Shift; CN is Case Normal. NLC indicates a **n**on-**l**egible **c**haracter (always suppressed by the line printer); and Ø represents the filler character, suppressed by all legible output devices in normal transfer modes.

The underline '_' does not advance the Flexowriter carriage and so is over-printed by the following character; it is used to represent Algol 60 'publication language' basic symbols, e.g., '**begin**', as '<u>begin</u>'.

The '$_{10}$' is the single-character exponent delimiter used in Algol 60's real number syntax.

Punched cards have 80 columns of 12 rows. Rows 0-9 represent the decimal digits with a single punched hole. The two rows above are called X and Y (11 and 12 on some non-KDF9 systems). Most non-numeric characters are represented by combinations of two or three holes, one from rows Y, X, or 0, and the others in rows 1 through 8.

APPENDIX 5: THE KDF9 GRAPH PLOTTER CODES

Effect:	Octal code:
no effect	00
step paper back	01
step paper forward	02
step pen right	04
step pen left	10
step pen right, step paper back	11
step pen left, step paper forward	12
lower pen	20
raise pen	40

All other 6-bit character codes represent invalid plotter commands.

REFERENCES AND BIBLIOGRAPHY

[Davis60] 'The English Electric KDF9 Computer System'; G.M. Davis,
BCS Computer Bulletin, Vol. 4 No. 3; 1960.

[EEC62a] 'KDF9: Basic Organisation of Controls; and Instruction Timings'; J.R. Lucking and J.P. O'Neil;
Report K/GD.y.73, The English Electric Company Limited; 9 July 1962.

[EEC62b] 'Notes on KDF 9 Main Control'; J.A. Edwards;
Report K/GD.u.293, The English Electric Company Limited; 7 September 1962.

[EEC62c] 'KDF 9—Transformer Sequence Units'; J.R. Lucking;
Report K/GD.y.81, The English Electric Company Limited; 15 November 1962.

For the above three documents, see: `http://sw.ccs.bcs.org/KDF9/Wichmann/techdoc.html`

[EEC69] *KDF9 Programming Manual*,
Publication 1002 mm (R) 2nd Edition, English Electric Computers Limited; October 1969.

[Findlay11a] 'The English Electric KDF9'; W. Findlay; 2011.

[Findlayxxx] 'The Software of the KDF9'; W. Findlay; (in preparation).

For the above three documents, see: `http://www.findlayw.plus.com/KDF9/`

[Haley62] 'The KDF9 Computer System'; A.C.D. Haley,
Proceedings of the Fall Joint Computer Conference, AFIPS Conference Proceedings, Vol.22; 1962.

[WR67] 'Competition for memory access in the KDF9'; C.S. Wallace and B.G. Rowswell,
Computer Journal, Vol.10, pp. 64-68; 1967.