THE D21 DATA PROCESSING SYSTEM BY SVENSKA AEROPLAN AKTIEBOLAGET, SWEDEN

BY

B. LANGEFORS

Reprinted from IEEE TRANSACTIONS
ON *ELECTRONIC COMPUTERS*
Volume EC-12, Number 5, December, 1963

PRINTED IN THE U.S.A.

The D21 Data Processing System by Svenska Aeroplan Aktiebolaget, Sweden*

B. LANGEFORS†

Summary—**The D21 data processing system design uses simple system structure and fast circuits plus flexible memory and is thus adapted to the use of advanced software and easy application to diverse fields. A survey is given of design objectives, background, and the hardware-software system.**

Design Objectives

IT APPEARS to be almost a general tendency among computer designers to strive toward the highest computing speed, reachable with the basic components available, or obtainable with the memory speed at hand. This has governed an almost continuous trend to more and more complicated computer design, in order to enable different kinds of time saving in internal processing. Thus the elegant simplicity, so basic to modern computer design, which is characteristic of the von Neuman computer layout, has gradually given way to various ingenious extensions in the form of index registers, multiple arithmetic registers and overlapping micro-operations.

In the design of D21 a decidedly different goal was set up. Thus, rather than attempting the very highest possible capacity, a medium-to-high capacity which was coupled with a large flexibility in memory size and terminal equipment was specified. The design should enable efficient handling of both administrative data processing problems and engineering computations. This was estimated to be most suitable layout for the Swedish Data Processing market and others as well, and the efficient application to both engineering and business data processing would make its application economical also for companies and organizations which would be too small to use a computer for one application only.

The explicit attempt at a great generality of application would also mean a good potentiality for introducing more scientific methods in business data processing which must be considered a very important possibility.

The design objectives as described strongly call for the use of advanced programming systems, and therefore, the computer should have properties which make the construction of compilers easy, reduce compilation time and enable economical use of compiled programs.

It will be seen that these design objectives could best

* Received August 27, 1963.
† Saab Aircraft Company, Linköping, Sweden.

be met by a system design, which means a definite step away from the current trend towards increasingly complicated hardware, mentioned above. It also means a decisive approach to a computing system consisting of hardware and software as an integrated whole, which, of course, also means a very close cooperation between hardware people, compiler people and applications specialists in the design work.

Background

The D21 system is developed on a background of experience in the use, since 1950, of several different computers for aircraft design and administrative data processing, as well as in designing and building single computers for use within the company. It may be of interest to describe very briefly this background before we discuss the design of D21.

After using the fast Swedish electronic computer Besk since 1953, and after some study of the market, Saab Aircraft Company decided in 1955 to build its own computer, later to be called Sara. Sara was built as a copy of the vacuum tube computer Besk modified to enable the attachment of magnetic tapes, punched-card equipment and printer. It has 2000 40-bit words in core store and 8000 words on drums; addtime is 54 μsec; multiply time is 350 μsec. Two instructions per word are used.

In 1957 Sara was put to work and in 1958 it was equipped with a magnetic tape system called Saraband. This was designed and built in our laboratory using redundant coding which permitted automatic correction of 1-bit errors and alarm for multiple-bit errors. It used $\frac{3}{4}$-inch tape with 12 channels of which 4 were used for information, 5 for correction, and 2 for block and word marking. The error correction system worked so well that it was decided to use 300 characters per inch rather than the 200 recommended as maximum by the tape-handler manufacturer.

In the design of Sara it was decided to have no index registers. The main argument was that only two binary positions per instruction (of 20 bits) were available, permitting a maximum number of 3 index registers, and this was considered far too small. Some special design features were proposed for Sara during its late design phase in 1956–1957 but were not adopted. Among these were a set of push-down stores. A single one of these stores would work as a large number of index registers for the special function of subroutine linkage, whereby a large number of subordinated subroutines, recursive or not, would be automatically handled.

Another push-down store with similar functioning was planned for looping and a third one for storing subroutine parameters (in the same way a push-down store is normally used today for Algol procedures).

Another modification planned was the introduction of an indirect addressing mode whereby all 20 bits of an indirectly addressed halfword would be used for the actual address. This would permit access to a maximum of one million words by indirect addressing and could be used for branching to subroutines anywhere in such a large store. For the working of these subroutines, a relative addressing with a span of 1000 instructions would then enable the procedures to work without being slowed down by indirect addressing in the majority of the instructions. For subroutine branching, a special jump instruction was proposed, which would store the return address in the first call of the subroutine where it was easily available by indirect addressing. The use of index registers for this function was thus replaced by a much better device.

While Sara was built the company carried on development of fast and compact transistor circuitry which was eventually to be used for an on-line computer for airplane control purposes. In this development a compact computer D2 was built and put to use for some special experiments in the fall of 1960. D2 is in many respects a prototype for D21.

This fact implied that the work on programming systems for D21 could be based upon a fairly broad and deep experience in systems programming and also that the needs of this, as well as the knowledge of its possibilities, could be properly used to influence the design.

Some Basic Design Considerations

The design goals requiring high efficiency and economy in the handling of business data, as well as engineering problems, at first sight appears to lead to severe contradictions and hence, bad compromises. The interesting fact is that, with the solution chosen, this has not turned out to be the case at all. In the eyes of many data processing experts (and perhaps all computer salesmen), the requirements on the computers for business data handling are completely different from those to be used for engineering computations or scientific work. A closer analysis, based on experience from both fields, shows that this is only partly true.

Tape files for business are often larger than those used for engineering. Instead, however, business data processing is typically associated with single scan of the tape files, whereas engineering problems in general call for complicated multiple tape scans. Matrix multiplication is a case in point. Consequently, both kinds of use call for efficient tape systems (or other backing stores).

It is likewise true for each main program pass that business data processing in general only needs to store a few records at a time from different files in the high-speed memory. However, the requirement of efficient tape handling calls for blocking several file records together when they are small. This is only possible when a fairly large memory is available. Further, in integrated data processing it is often possible to eliminate several file passes by performing several processes at the same passage. This is only feasible if the memory is large enough to store simultaneously all (or most of) the programs for those processes. Both applications thus can use large memory.

It is true that in engineering computations many more arithmetic operations with higher precision (that is, more significant figures) are needed than in typical business operations. Therefore, the use of data packing to increase effective tape speed, although of value in itself, is dispensed with in engineering or scientific computers because of the consequential slowdown of the arithmetic operation, even in the case of hardware implementation.

Here is a difference which will easily be of consequence for the computer design. In fact, this will be the case when the basic speed of the circuits used is not in itself high enough for the desired over-all capacity. The "scientific computer" will need some extra "arithmetic circuitry" such as longer arithmetic registers (at least 40 bits), built-in floating point arithmetic and a significant number of index registers. The "business computer" will instead use "data shuffling circuitry" such as one-character words with serial processing of characters generally, but erroneously or at least misleadingly, called "variable word length" features.

A computer of general applicability would then have to be provided with two different "*categorical packages of circuitry*," the "*arithmetic*" *package* and the "*data shuffling*" *package*. For each application one of them would always be unused. In addition to this is the fact that the two categorical packages are somewhat contradictory, which adds complexity (and cost) and tends to reduce speed.

These difficulties are, however, eliminated if the desired capacity is obtained by sufficiently high speed in the basic circuits together with sufficient memory, instead of by the additional packages of categorical circuitry. The higher cost for faster circuits may well be balanced by the elimination of the additional special packages otherwise necessary. Whether this is the case or not will, of course, depend on the status of the hardware technology in relation to the capacity requirements. This of course will vary with time. When high speed eliminates the need for categorical circuitry, then this difference between "scientific computers" and "business computers" disappears.

If the circuits are fast enough, then it will be possible to eliminate not only the categorical circuitries, and with them the differences, but also other special circuitry which is used to increase capacity for all types of applications, thus bringing still more saving of cost and a gain in simplicity. Examples of such special circuitry are index registers, multiple accumulators or comparing registers, electronic switches and a set of special instructions.

The several eliminations of categorical circuitry and special circuitry, shown to be made possible by the use of fast circuits, will themselves bring about an additional saving in that no positions in the instructions are needed for indexing or tagging, and fewer positions are needed for the "function part" (or "operation" part) of each instruction.

This, on the other hand, will be counteracted by the need for a higher number of instructions in a given program. This effect, however, in its turn can be counteracted by the use of powerful macroinstruction software, a point we will discuss more below.

The effect of the extra circuitries, however, is threefold:

1) Increase of computing speed or capacity;
2) Reduction of the number of stored machine instructions;
3) Simplified machine coding.

Thus, to justify the elimination of extra circuitry, we not only have to compensate for lost capacity by memory and circuit speed but we also have to balance the increased number of machine instructions and, most important, balance more detailed machine coding by providing good software.

It is easy to show that this can be done better with the use of increased speed instead of extra circuitry. It has turned out, also, that advanced software will reduce the number of stored instructions when the problems processed are not very small.

Past experience indicated that one main obstacle to using advanced software was its inefficient utilization of special (or categorical) circuitry in the object programs. From the very important software point of view, it would, therefore, be a great help to have a computer which did not depend on special circuitry for its capacity. Advanced programming systems then would not lead to lost machine efficiency; compilation would be fast and compilers easier to make.

The emphasis on software to replace hardware calls for efficient handling of subroutine linkage. Even this was found to be obtainable in a very efficient way without needing extra hardware.

It is not only the internal operations of the computer which can be made simpler by the use of faster circuits; the functioning of the terminal equipment can be simplified by the program-interrupt system made possible by speed. Again, in this connection, speed can also make it feasible to solve some of the problems by means of software.

The Choice of a Binary System with 24-Bit Words

Whether to use fixed word machines or so-called variable wordlength is a subject that is constantly debated. The author would prefer to pose the question otherwise: which is the optimum balance of serial- and parallel-mode functioning and of binary vs decimal arithmetic, as regards over-all speed of operation and accessibility to variable length data?

Given the basic circuits, the highest arithmetic speed is obviously obtained when use is made of parallel operation on so many binary positions as are needed to store the maximum-size numbers to be worked. If these numbers are represented in binary again, this operation will be faster than if represented in decimal. Such long-

word arithmetic units will, on the other hand, call for extensive, and thus expensive, circuitry. Further, to extract short data packed into long words will be slower, the longer the words are, unless extra circuitry for part-word operations is provided which is then a cost and a complication.

If, on the other hand, parallel operation is used on fewer bits, that is, if shorter words are used, then the arithmetic unit becomes smaller and less expensive, and short data are more easily extracted. At the same time, handling longer data is still easy because multiple word operations are simple, whether handled by hardware, as in so-called variable wordlength machinery, or by program loops. In either case the number of words and hence the item length can easily be allowed to be variable.

From this point of view, the so-called variable word-length machines or character-based machines are actually computers with a fixed word length of six to eight bits. They are then provided with automatic multiword operation controlled by some sort of item separator.

In the design of D21, the choice of a wordlength of one character size, *i.e.*, 6 or 8 bits, was not considered the optimum compromise and neither was a long word of 40 or 50 bits. Further, with the character-based machines, it is practically impossible to work with items of smaller size than an alphanumeric character; that is, no computer has such a rigidly fixed wordlength as the so-called variable word-length machines. Thus, already the storage of numeric data, the most common type, is rather inefficiently handled in such a system, and this is still more true for the not uncommon items which require only one, 2 or 3 bits.

The compromise chosen for D21 is, instead, to have a wordlength of 24 bits in parallel mode and to use binary arithmetic. This gives sufficient precision for most arithmetic in business operations and also for several types of scientific computations which are then handled with very high speed. Further, this high speed is almost retained in comparing larger numbers, because with this wordlength only one single subtraction will be sufficient in the majority of operations. Further, for the majority of engineering computations where higher precision is necessary, double precision will mostly be sufficient and can still be obtained with fairly high speed with fast circuits. This wordlength also is sufficient for all index computations needed in using the extremely convenient device of subscripting variables in advanced programming systems like Algol. Thus, while this wordlength is high enough to permit fast arithmetic operations, it is also short enough to permit long items such as alphanumeric names to be handled as multiple-word items, enabling easy handling of dynamically varying length, using any type of item separator for control. Further, this wordlength is short enough to permit very fast extracting of short items by means of shifting, and this can be used for itemlengths down to 1 bit, contrary to character-base design.

Further, this wordlength is long enough to store one machine instruction, and still short enough to permit the storing of one address in one word when multiaddress macroinstructions are used. Finally, this wordlength is short enough so that we can afford to work with word-synchronized data during the execution of all macroinstructions, so that packed data occur only in input and output areas of the memory. Of great advantage, especially for business application, is the ease with which binary systems can accept input data with varying codes.

Description of the D21 Computing System

It should be obvious from the discussion given above that it does not make sense to look at a computer such as D21 by viewing the hardware only. Rather, the computing capacity and the ease of use is completely dependent on the combined system of hardware and software, and only when thus regarded can its properties be judged. The existence or absence of some hardware details so often disputed have no relevance for the user. Hence, such disputes are meaningless. The resulting speed, capacity, ease and safety of use are, of course, the only factors to consider.

We describe the D21 system by giving first a brief review of its main hardware layout with special mention of details which influence the software adaptation on the applicational systems. Then we give a description of the functioning and the repertoire of instructions, not of the hardware alone, which would not make sense, but of the combination hardware-basic assembly system DAC. Finally, we describe the problem-oriented software.

D21 Hardware System

Instruction Word Structure

The instruction word comprises 24 binary positions (or bits) numbered 0–23. Positions 0–5 contain the operation part or function part; 6 and 7 are so-called marking positions or tags, and the remaining sixteen positions contain the address part of the instruction, see Fig. 1.

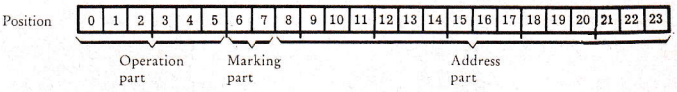


Fig. 1.

The first one of the address bits indicates memory if it is zero and a terminal unit if it is one. Thus, terminal units are addressed in exactly the same fashion as memory cells so no special input or output instructions are needed. The remaining 15 address bits permit addressing 32,768 memory cells and the same number of terminal units. The two marking bits have the following function:

0 0 Regular functioning
0 1 Reserve
1 0 Normal indirect addressing
1 1 Indirect addressing with "step ahead".

Over-all Structure of D21

The over-all structure is shown by the diagram, Fig. 2. The computer works in 24-bit parallel modes with one-address instructions. Some instructions work with 47-bit numbers by built-in double precision microprograms. Negative numbers are represented with 2-complement and the arithmetic is binary, with a fixed point adjacent to the sign position. The internal registers communicate via the internal bus line. This has a capacity of 1.25×10^6 24-bit words/sec, *i.e.*, the transfer of a number from one register to another is made in 0.8 μsec. The memory size is from 4096 words up to 32,768 words in modules of 4096. It is of coincident current ferrite-core type. The total memory cycle time, including address modification, is 4.8 μsec. All modules use the same common communication circuits which keep the marginal cost for memory augmentation fairly low. Protection circuits are provided which prevent power failure or regular shut-down of the computer from producing changes in memory contents.

The arithmetic unit contains, among other things, an *accumulator register* (AR) and a "*multiplication register*" (MR), both with 24 binary positions, 0–23. In addition to this, AR contains one position 00: *the overflow position*. These can also work as one long register of 47 bits, rather than 48, because the "sign position" of MR is void in this case. This long register is used by some "long instructions."

The control unit works with an internal clock of 2.5 Mc frequency. It utilizes a *micro-order generator* (MG) which translates each order code into a sequence of pulses (the microprogram) controlling the micro-operations. Changes or additions to the instruction repertoire are easily introduced by changes in MG. Of interest to the user is the *address register for operations* (AsR-O). This a "counting register" which can add +1 into position 23. It has fifteen positions, 9–23.

The memory unit also contains two counting registers capable of adding +1 or −1 to position 23. One is the 15-bit *address register for data* (AsR-M) with positions 9–23. The other is the 24-bit memory register (MsR).

The memory register is used for all communication with memory cells. Any number or instruction which is read is written again into the same cell. In some cases (*step-ahead functions*), +1 is added to position 23 of MsR before writing back.

The memory register can be used for program control by connection to the *sign indicator* (I) which is a 1-bit register connected to position O of MsR (and sometimes to position 9). This program control is made possible by being associated with one branching instruction "I GO TO". When "I GO TO, A"; occurs in a program, branching is performed if I contains a 1 and not if I contains O. The sign indicator I is set to 1 when a number of negative sign is read or written in memory. It remains until one of several instructions affecting I has occurred.

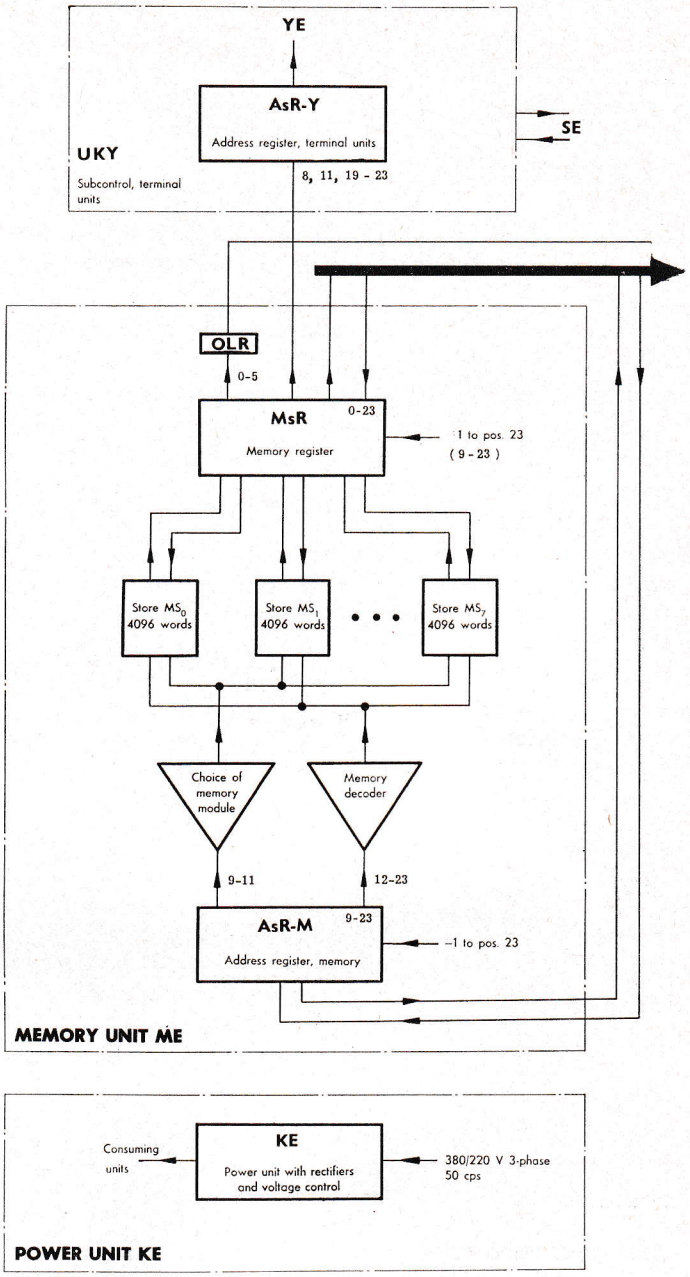


Fig. 2.

By means of I and the function I GO TO, program control is obtained without influencing the contents of AR or MR. I can also be used for control of program loops. In this case, I is connected to position 9 rather than O, so that it will indicate the sign of the address part of the word. It is thus used in connection with a count instruction "+1, A;" which adds +1 to the memory cell A and then brings position 9 to I.

Communication with terminal units is via the *external bus line* (YB). This is connected to the internal bus line by means of the *external bus register* (YBR) of 24 bits. From 32 input and 32 output, channels can be used.

When an instruction to be executed contains a unit in position 8, thus addressing a terminal unit, the positions 8, 11 and 19–23 in MsR are transferred to the

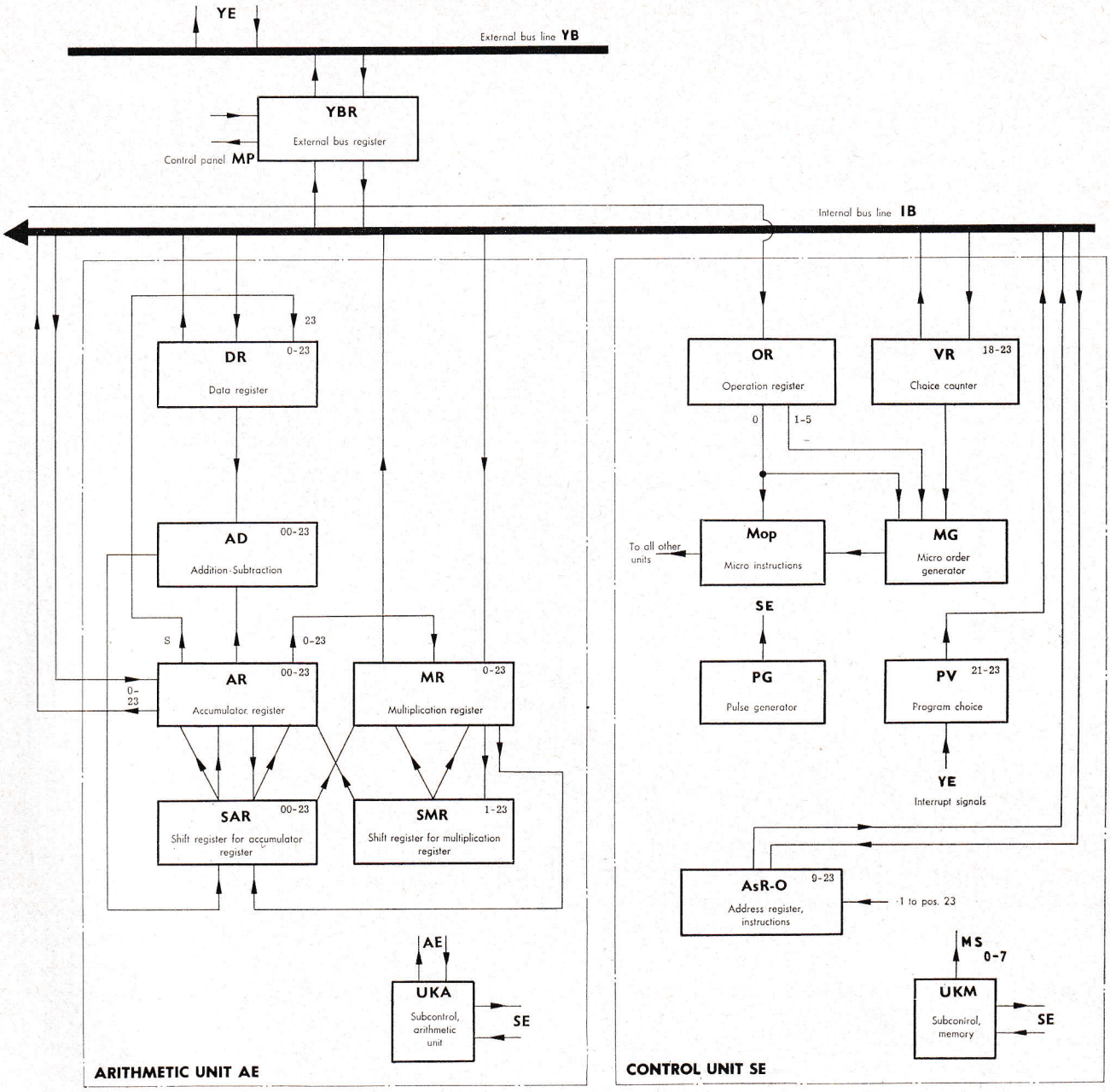


Fig. 2 (*Cont'd.*)

address register for terminal units (AsR-Y), a part of the *subcontrol for terminal units*. The latter interrogates the addressed terminal unit whether it is ready or not. If it is not ready, the computer waits for a *ready signal* from the terminal unit. "Interrupt signals" which occur during this waiting will, however, cause the waiting instruction to be regarded as not started. If ready, the terminal unit will perform the ordered communication with the arithmetic unit *via* YBR.

YBR is needed as a buffer between the internal bus line and the external bus line because the latter is considerably slower. This enables communication over longer distances when terminal units are concerned.

An *interrupt system* is provided to permit simultaneous operation of different terminal units and of the computer, without need for buffer storage. Two different interrupt signals are provided. One causes program branching (with storage of return address) to a fixed memory cell. This initiates "interrupt control software." The other one causes a break without branching while one word is transmitted between the terminal unit and a specified memory cell, after this cell specification has been changed by $+1$; *i.e.* so subsequent cells will be used for subsequent transfers (step ahead).

The external bus line and the interrupt system make the connection of different terminal units easy. Any such unit only has to generate an interrupt and a ready signal, react to a start signal, and be provided with one register of 24 bits or less which can communicate with YBR.

The Magnetic Tape System

Magnetic tape units are connected to the computer by means of a special tape communication unit (TCU). This can connect up to eight tape handlers, but several tape communication units can be attached.

By word, TCU transfers information between tapes and the computer word, using a 24-bit register. Any kind of data packing on tapes is obtained by packing these data in words in the memory used for tape communication. Information is stored on tape in blocks of variable length, from 1 to 1023 words. 1-inch tape is used, providing 16 tracks of which 15 are used. 8 tracks carry information so that three transverse rows on tape carry one computer word. 5 tracks are used for redundant information to correct automatically all 1-bit errors on reading and to check for multibit errors on reading and block counting. 2 tracks are used to indicate block starts and block ends by 2 different combinations. These are also checked and corrected. The following operations can take place simultaneously:

1) Reading one tape and writing another;
2) Reading one tape and block passage while counting blocks on another;
3) Writing and block counting;
4) Block counting on 2 tapes.

Interblock gaps are about 1 inch which corresponds to about 100 words. Recording density is 300 rows per inch. "Interrupt with branching" is associated with the start and finish of block communication, whereas "interrupt without branching" is initiated by the 24-bit register of TCU each time it is filled or emptied. Thus, subsequent words in memory are loaded during tape reading at instants when tape rows have been transferred to fill the TCU register. Writing is analogous. Other activity can go on during this process except for the short intervals of interrupts and transfer between the TCU register and YBR.

Rather than possible alternative solutions, the choice of the error correcting and detecting system used was based on the extremely favorable experience with such a system on the earlier computer Sara. Additional arguments follow.

The relatively high recording density used in modern tape systems (300 rows per inch in our case) bring some consequences which are very often overlooked. The distance between rows, that is, the longitudinal distance, is 1/300 inch, while the transverse distance, between tracks, is more than 10 times greater. Therefore, a disturbance in the magnetic communication between the tape and the head which is large enough to disturb 1-bit position, will disturb several bits in the longitudinal direction along the same track. This almost completely eliminates the value of longitudinal parity checking which would otherwise have been of interest. Some published mathematical analyses of longitudinal checking disregard this fact. Their results are accordingly unrealistic by no small measure. Thus any time there is a bit error in a row, the corresponding track will have a multibit error. Only in 50 per cent of the cases would this have an odd number and be detected by longitudinal parity checking. Similarly, a 2-bit error in a row will introduce multibit errors in two tracks, and in 25 per cent of the cases both tracks will have even numbers of errors which then go undetected.

While dense recording thus makes the longitudinal checking almost useless, it also makes it unnecessary. Thus a 2-bit error in one row will be followed by 2-bit errors in the neighboring rows because of the short longitudinal distance in relation to track width. In some row or rows, then, one track will still be disturbed while the other is not. The result is one or more rows with 1-bit errors in the vicinity of all (or almost all) 2-bit errors. Only when the disturbances on two tracks have exactly the same extension to the precision of less than one tenth of a track width in both longitudinal directions, would this not occur. An analogous situation occurs with multibit errors which will therefore almost always be surrounded by errors of every lower multiplicity. Therefore error detection is only required for 2-bit errors when 1-bit correction is used (and for $n+1$-bit errors when n-bit errors are corrected).

With the system used, all 1-bit errors, whether generated during writing or reading, are without significance, although they can be counted to check tape system condition, while all multibit errors occurred during writing or reading will lead to error alarm during reading. Most of them are then eliminated by rereading. (In the Sara system about one error alarm per week is obtained, but only one to ten per year are unreadable.) No errors go undetected. This is true for most modern tape systems. Therefore, the significant figure of merit for comparison of different checking principles for tape systems is the frequency of occurrences of nonreadable data. This can be reduced by automatic correction, as used in Sara and D21, for instance, or by automatic detection at an instant when corrective action can be automatically initiated, or both; for instance, by rereading during tape read and by check reading during tape write. This can be computed in a simple way, whenever the basic error frequencies are available from statistics. To see this we first introduce some notations.

Let

w^1 = frequency of 1-bit errors generated during writing;
w^m = same for multiple bit errors;
r^1 and r^m be analogously defined for errors generated during reading or between writing and reading;
d^1 and d^m be analogous frequencies caused by permanent damages on the tape.

Thus, the total number of errors introduced on the ref-

erence quantity of tape data, from the instant of output to tape through the event of being input on a subsequent tape read, is $w^1+w^m+r^1+r^m+d^1+d^m$. However, some errors generated after writing are nonconsistent in the sense that they can be corrected by iterated reading. As this can always be initiated at read time, in systems with error detection, we have only to consider the smaller number of errors obtained when nonconsistent errors are eliminated.

Let

$c^1(<1)$ be the "consistency factor of 1-bit read errors," defined by $c^1 \cdot r^1 =$ frequency of consistent 1-bit read errors

$c^m(<1)$ be analogously defined for multiple-bit errors.

The total error frequency thus will be (for any system with error detection)

$$t = w^1 + w^m + c^1r^1 + c^mr^m + d^1 + d^m.$$

We are interested in different methods for reducing t. We consider 1-bit error correction and, alternatively, check reading after writing.

The total frequency of occurrencies of nonreadable data when 1-bit error correction is used is seen to be

$$t^1 = w^m + c^mr^m + d^m$$

whereas the check reading gives

$$t^c = c^1r^1 + c^mr^m$$

if we also make the assumption that the check reading is also used to skip damaged portions of tape during writing.

A combination of both methods gives

$$t^{1\cup c} = c^mr^m.$$

A quantitative comparison of these three quantities is necessary to enable an economical choice of design. This calls for numeric values for the basic frequencies entering the formulas. The experience from Sara indicated roughly the following relations:

$$w^1 = c^1r^1$$

$$w^m = c^mr^m$$

$$w^1 = 1000\, w^m$$

$$c^1r^1 > 1000\, r^m.$$

The first two relations are explained by the fact that writing is done with multiply overpowered write signals with the result that smaller physical disturbances have no effect on writing while causing errors in reading. The remaining relations are statistically natural, and in the first approximation one would have $w^m \approx (w^1)^2$ and similar for r^1, r^m. This would have given a factor of 10^7 instead of the (conservative) 10^3, above.

With these data we obtain, for undamaged tape, $t^1 = 2w^m$,

$$t^c > 1000w^m, \qquad t^{1\cup c} = w^m.$$

Thus, when damage is rare and damaged tapes are taken out of use, then 1-bit error correction is vastly superior to check reading after writing while the combination of both give only slight improvement. If tape damage is expected often, then check reading might be desirable. The experience from Sara did not indicate that such would be the case.

Other Terminal Equipment

There are, of course, a set of other types of terminal equipment that can be connected. In fact, the design is especially made such that the introduction of any such equipment is easy at any subsequent time.

Presently in use in every installation are a control desk, a decimal display register and manual program switches. Further, a punched tape reader (1000 characters/sec), tape punch (150 characters/sec), card reader (800 cards/min), card punch (120 cards/min), and line printer (900 lines/min) are optional, as well as magnetic tapes and digital-to-analog and analog-to-digital converters for on-line operation of $x-y$ recorders, or in connection with process-control applications.

The Combined Hardware-Software System, DAC

The basic assembly system DAC contains instructions for all machine-built operations as well as many others. It is therefore most convenient to give a review of the machine functioning in DAC language rather than in machine coding, the more so as DAC uses a very natural and rather machine-independent symbology. Finally the whole DAC system, rather than its "machine-built subset," depicts the properties of the D21 system, and from the programmer's point of view, it makes no difference whether a DAC instruction is a machine operation or a macroinstruction as soon as he has knowledge of the operation time. We thus start by giving a brief description of the DAC language structure.

A DAC instruction, like a machine instruction, is composed of an *operations part* and an *address part*. DAC uses one-address instructions but, as we shall see, subroutine calls are so simple that they look almost like single instructions. Subroutines may, however, use any number of parameters so that the subroutine calls actually introduce multiaddress instructions of variable length into the system.

It was considered very important to be able to use short names for frequent and simple concepts and long, descriptive names for infrequent, complicated ones. This holds for operations as well as for data. DAC therefore, unlike most symbolic assembly systems, uses variable-length operations parts and address parts of instructions. That is, names of operations and data may have any length. Therefore terminator symbols have to be used, and in DAC every operation part is terminated by

a comma, and the address part (as well as the instruction itself) is terminated by a semicolon.

As engineers, mathematicians and, in fact, all people who have learned to make use of the power of concise symbology use many one-character symbols, a large alphabet, also containing minors and some special signs, is used in DAC. If, for instance, i and I are used as symbols for different objects in a formula, it is not true that the assembly system permits use of original data names if only majors are permitted. To change some symbols may seem trivial but experience proves it to be very awkward indeed.

The address parts of DAC instructions can be data names or numeric values; indirect addressing is obtained if address parts are enclosed in parentheses. Thus, op, (A); is equivalent to op, a; (a = address part of content of A), and +=, A; adds ar to a and then stores in A. Likewise C+, A; , clears AR (indicated by C) and then adds a to AR while =C, A; stores ar in A and then clears AR.

Prefixes L, D and F are used as *categorizers*, *i.e.*, to indicate that an operation is of the category *long* (L), *double precision* (D), and floating point (F), respectively. L*, A; forms the 47-bit product of ar and a in ARMR while D*, A; forms the product (likewise 47 bits) of armr and a, a+1 and F*, A; finally multiples in floating point (two D21 words per number).

When another register than AR (or its counterpart ARMR in L and D operations) is to be used, it is named in the operations part. Thus C+MR, A; brings a into MR and I GOTO indicates a jump controlled by the status of the indicator register I.

TABLE I

ARITHMETIC OPERATIONS, BASIC CATEGORY

Equation Number		Basic Operation	Time (μsec)	Combination Available with Preceding Operation	Combination Available with Succeeding Operation	Variants for Registers, other than AR
1	+, A;	ar+a to AR	9.6	C,	=, =C,	C+MR,
2	−, A;	ar−a to AR	9.6	C,		
3	*, A;	ar·a to AR, rounded	35.2–40.8			
4	/, A;	ar/a to AR	46.4			
5	=, A;	ar to A and AR	9.6	+,	C,	MR=, MR=C,
6	P, A;	effective address to MR	9.6			
7	∧, A;	ar∧a to AR (extract)	9.6			
8	←, N;	ar left shift N places	7.2+(N−4)0.8			
9	→, N;	ar right shift N places	7.2+(N−4)0.8	C,		
10	NORM, A;	ar normalized, no of shift to AR	10.4+n·0.8			
11	+1, A;	a+1 (integer) to A pos. 9–23 and sign to indicator I	10.4			
	ABS	/armr/ to AR	6.4 or 11.2	−1		
12	math. f functions and editing operations					

Names in minors indicate content of register with same name in majors. For instance, the first row indicates that +, C+, +=, +=C, C+MR are available variants of +.

of A). Brackets are used to indicate the step-ahead mode of indirect addressing; op [A]; is equivalent to op, a+1;. Indirect addressing adds 4.8 μsec to the operation times (in the stepping mode 5.6 μsec).

It also makes it possible to use DAC operations part language for other computers in a way that will be intelligible when one knows the DAC operations part language syntax. It can thus also be used to describe the orders of any computer.

The operations parts of DAC instructions may be a combination of more elementary ones. Thus +, A; adds a (the content of memory cell A) to ar (the content of the accumulator register AR), and =, A; stores ar in

Arithmetic Categorizers:

F	floating point, 40-bit mantissa
D	double length, 47 bit
L	partly double length
F, D	are applicable to most of operations 1, 2, 3, 4, 5 and 12, SF to 12
L	is applicable to 3, 4, 8, 9, 10. For instance L→, N; shifts ARMR N steps to the right, so that the N right-most positions of AR are moved to MR. The sign of AR is conserved. Instead LC→ will shift ARMR right with 0's fed into AR_0

and

AS	operate on positions 8–23 only
MAS	operate on positions 6–23 only
LEFT	operate on positions 0–11 only
RIGHT	operate on positions 12–23 only

which apply only to 5.

Some examples of operation times for category variants may serve to illustrate the speed available.

D+	16 μsec
D*	300 μsec
D/	445 μsec
L*	35.2–40.8 μsec
L←	7, 2+0, 8 (n−4) μsec
F+	265 μsec
F*	430 μsec
SQRT (square root)	410 μsec
DSQRT	590 μsec
FSQRT	565 μsec
SFSQRT	395 μsec
SIN (sinus)	340 μsec
DSIN	1505 μsec
FSIN	1865 μsec
SFSIN	530 μsec.

Note that for the mathematical functions, of which only a few have been shown, floating point arithmetic is as fast as fixed point arithmetic. The remaining categorizers do not affect time.

We show later some editing and string handling (*i.e.*, text handling) operations available in DAC, as well as variable-length data handling.

Program Control Instructions: In basic form, these are

```
GOTO, A;  branch to A; 4.8 μsec
DO    , A; a is used as instruction, 4.8 μsec+instruction time
STOP, A;  stop, on restart branch to A
HOLD;     when all peripheral units have completed specified
          operations continue with the next instruction.
```

Categorization is merely for GO TO which also permits a successor operation PR. Therefore, we list the categorized operations in complete form.

```
+GOTO, A;      branch to A if ar≥0
−GOTO, A;      branch to A if ar<0
IF ZERO, A;    branch to the second next instruction if ar≠0
IGOTO, A;      branch to A if i=1 (indicator neg.)
OVGOTO, A;     branch to A if overflow
NOVGOTO, A;    branch to A if not overflow.
```

Time for most branches is 4.8 when performed, 6.4 otherwise, and the combined operation.

```
GOTO PR (or PR), A;   If this instruction is labelled L then L+1
                      is transferred to A's address part followed
                      by branching to A+1. AR and MR not
                      affected.
```

I GOTO, is used together with +1, for simple loop programming. Thus the following set of instructions will loop N times:

```
        C−    , N;     Bring −N
        =     , Loop;  into Loop
Proc:   −
        −
        −
        +1    , Loop;  Content of Loop is reduced by 1.
        IGOTO , Proc;  The first N−1 times this statement is
                       seen. Loop is negative, I is one, branch to
                       Proc is made. The Nth time Loop is zero
                       and I=0 and no branch is made.
        —
        —
        —
```

The package of instructions, PR, P, DO, together with the indirect addressing modes, make calls of even complicated multiparameter subroutines very simple and fast.

A simple example will exhibit this. Suppose we want to use a subroutine called Prac which uses two parameters whose values then must be transferred to Prac before or during its execution. Let these parameters be named Length and Width. The instruction in the main program which calls for Prac may be labelled L. The relevant part of the program will then be

```
        .
        .
        .
L:      PR, Prac;
        P  , Length;   } Call for subroutine Prac.
        P  , Width;
        —
        —
        —
```

It is seen that the call for Prac is nothing but a two-address instruction of a somewhat modified form.

This simplicity for the programmer is not bought by extensive and time-consuming operations in the subroutine. We show this. The first cell of the subroutine is labelled with its name. Let us assume that the subroutine starts by retrieving its two input parameters and stores them in Le and Wi. After the branching of PR, the relevant parts of Prac will now be

```
Prac:   , L+1;
        DO, (Prac); MR=, Le;
        DO, [Prac]; MR=, Wi;
        —
        —
        —
        GOTO, [Prac];
```

The instruction L: PR, Prac; in the main program brings L+1 into the first cell in Prac. and branches to the succeeding cell which contains the first instruction of Prac.

This says: DO, (Prac); where parentheses indicate indirect addressing so that the operation performed is DO, L+1;, because L+1 is the content of Prac. Now DO, L+1; is equivalent to P, Length; which is stored in L+1. This brings the address of Length to MR. Then control goes to the next instruction of Prac, *i.e.*, MR=, Le; so that the address of the Length value is now stored in Le. The following instruction says DO, [Prac];, this time indicating indirect addressing with step-ahead. Thus +1 is first added to Prac, whereupon it is used for an address. This makes the instruction work as DO, L+2; that is P, Width; which then is brought by MR=, Wi; to Wi. The last instruction in Prac is GOTO, [Prac];. This is equivalent to GOTO, L+3;, provided no step ahead has been done on Prac in the program which was left out of description. Thus, processing is guided back to the correct place in the main program.

Handling of Strings (Text Lines) and Word Groups

In DAC a string is a sequence of characters. Rather than 6 bits, 7 bits are used to permit handling also minor letters. Strings are stored in DAC with 3 characters in the last 21 bits of a word. The first bit indicates whether the next word belongs to the string also. If not, then the remaining two bits indicate how many characters are in the word. Thus string length may vary dynamically.

The DAC instruction *MOVE STR, A; P, B;* moves string A to B, and indicates the number of words of A in AR, and the number of characters in MR. *FROM STR, A; P, N;* brings the Nth character in A (as a string) to AR. If impossible, the instruction makes AR negative. *TO STR, A;* puts a character in AR to the end of string A. *STR EQU, A; P, B;* compares the strings A and B. Then ar ≥ 0; is set, if equal; if unequal, ar < 0. At unequal, the ordinal number of the first unequal word is put into MR. Some instructions affect groups of words: *MOVE, A; P, B; P, N;* moves N words starting at A to B and its N $-$ 1 successors. *ZERO, A; P, N;* puts zeros into A and N $-$ 1 successor words.

Conversion from integer to string combined with editing is ordered by *INT TO STR, A; P, RHHDD; ERROR:* $\cdots$; "R" in RHHDD controls editing, for instance, zero suppress, floating sign, an s.f. "HH" number of integers, "DD" decimals in the string obtained. If the number converted gives a number of digits greater than indicated, branching goes to "ERROR," otherwise to the cell after. *STR TO INT, A; ERROR,* $\cdots$; is analogous.

Input and Output Instructions

We give a few DAC instructions to illustrate. *OPEN CARD IN, A1; P, A2;* assigns buffer areas A1 and A2 for card input. A1 = A2 gives one single area. *READ CARD;* reads a card to the available area A1 or A2. *GET CARD STR, A; P, BBBNN;* NN card columns, starting at BBB are moved to A as a string; information is put into AR about address for character BBB and whether the card field was blank, numeric, numeric with some signs, alphabetic, and whether BBB + NN > 81 (for 80 column cards). Card output is analogous. Printing is handled with the instructions *PRINT, N;* and *ADVANCE, N;* and *PUT PR STR, A; P, BBBNN;* which, respectively, prints a line followed by paper advance controlled by N, advances paper, and puts the string starting at A into the buffer area for printing with first character in BBB hammer position and checks whether BBB + NN > 121 (for 120 position printer).

Handling of Nonuniform Length Data

Both business and scientific applications encounter data of nonuniform length. To store data, which are much shorter than the memory words, in separate words wastes space on tape and in memory. Packing data, so that each takes only as much space as it needs, therefore reduces tape space and increases tape speed.

The specific features of D21 make feasible a solution to the variable-length data problem which does not reduce internal processing speed. This is obtained by performing data packing on output to tape and unpacking on input. No "word marks" or item separators need be stored on tape, except for dynamically varying data lengths; thus, in this system, tape handling is on packed data, giving both speed and space advantages. Instead, internal processing is on word synchronized data giving maximum speed but requiring extra space. The latter is kept small because word length is small. In fact for scientific applications, where in general much more data need be in memory than in business applications, the data, also when word-synchronized, will require fewer binary memory positions than in normal types of scientific computers, because short data take up only 24 bits rather than 40 to 60.

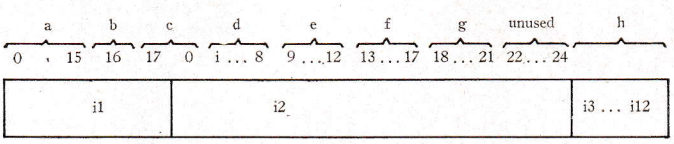
In addition to this, however, data size information and packing and unpacking routines also have to be stored. Part of this is balanced, on comparison, by word marks used in other systems.

In this system for handling variable length data in a way which permits both maximum tape handling speed and maximum internal processing speed, an intermediate additional processing has to take place, *e.g.* the packing or unpacking of data. This operation, of course, takes some time.

A further advantage of the method of doing packing and unpacking as an intermediate between input and output, and thus for whole records at a time, is that unpacking is simplified and speeded up, and packing still more so, when done in sequence. In addition, this operation prepares records for processing or output so that no extra handling has to be done to this end, which is otherwise the case as soon as records are grouped, several into one tape block. This mode of working also is made easier and faster by the indirect addressing facilities. In this process the data can be packed completely so that no considerations has to be given to word boundaries and so that data space can go down to one bit when this is sufficient.

In order to give the reader an idea of the packing efficiency and the speed obtained with this system we take a simple example. We assume that we have a sequence of data terms which are denoted below by a through h for short, together with their "natural" names

			Number of bits
a	Part number	0–50000	16
b	Variant code	1–2	1
c	Quantity	1–200	8
d	Year	00–99	7
e	Month	1–12	4
f	Day	1–31	5
g	Name length	1–8	4
h	Part name	1–30 alphanumeric characters	1–8 words



and indication of their minimum and maximum numerical values and the number of bits required for storing the maximum value. The last term "g" or "Part name" is assumed to require 1–30 alphanumeric characters of

6 bits each. This has been regarded as sufficient for justifying the assignment of only so much space (on tape) as is needed for the actual value, thus varying from one occurrence to the next one. This in turn requires length information to be carried with each occurrence. In the example, it is supposed to be stored as the term "Name length." The dynamically varying term is supposed to be assigned an integral number of words for simplicity.

We give below a DAC program which will move the terms to be unpacked in a sequence of "work words." Two cells called Dataword and Workword, respectively, are used to hold the current address of the dataword of packed data and of unpacked data, respectively. This information is thus available by indirect addressing to these words.

```
                                    address for workword for a = wa
        C+,     wa
        =,      Workword
        C+,     i1;
        =,      Dataword;
14.4 μsec C+,   (Dataword);  word i1 to AR
12        LC→,  9;           9 right shifts, O's inserted at left
                               side of AR
14.4 μsec =C,   (Workword);  term a unpacked in wa
 7.2 μsec L←,   1;           one bit is shifted left from MR to
                               AR
15.2 μsec =C,   [Workword];  term b unpacked in wb
10.4 μsec L←,   8;
15.2 μsec =C,   [Workword];  1st part of term c to wc
15.2 μsec C+,   [Dataword];  word i2 to AR
23.2 μsec LC→,  23
18.4 μsec +=C,  (Workword);  complete term c unpacked in wc
 9.6 μsec L←,   7;
15.2 μsec =C,   [Workword];  term d unpacked in wd
 7.2 μsec L←,   4;
15.2 μsec =C,   [Workword];  term e unpacked in we
 8.0 μsec L←,   5;
15.2 μsec =C,   [Workword];  term f unpacked in wf
        L←,     4;
        =C,     [Workword];  term g unpacked in wg
        C−,     (Workword);  −term g to AR
        =C,     Loops;       −Name length in Loops
   L1:  C+,     [Dataword];  Part name moved from datawords
        =C,     [Workword];  of number = value of Name length
        +1,     Loops;       to data words of number equal to
        IGOTO,  L1;          maximum value of Name length
                               (8 in this case).
        —
        —
```

We have written the execution times to the left of some of the instructions. Note that terms a through f correspond to 15 characters. They occupy 2 words minus the space taken up by g; that is, they occupy 45 bits. Thus, on the average 3 bits per character are used. In a character-based storage system, they would have taken up 15×6 bits plus 6 word marks of 6 bits making a total of 126 bits. Alternatively, 15×7 bits would have been needed, with one bit per character reserved for word marking, that is, 105 bits in total.

For the part name, we notice that we have stored 4 alphanumeric characters per word but reserved an integral number of words. In the last word we therefore have 3, 2, 1, 0 characters wasted alternatively and with equal probability, or $6/4 = 1.5$ characters or 9 bits, on the average. In addition, 3 bits were spent for the name length information. Thus we have in total 12 bits unused for the alphanumeric term, as compared with 6 bits for a wordmark character or, alternatively, as compared with 1–30 bits for one-bit wordmark position in each character position of other systems.

We must take into consideration, however, that this way of handling the alphanumeric term as a dynamically variable one, that is, on tape but with fixed length in memory, makes its further processing so simple that one can afford to use records with several dynamically varying term lengths. This will correspond to saving several words on the average for each occurrence of this term in the present system as compared with systems not using the dynamic variability.

The time taken for unpacking the terms a through f amounts to 216 μsec which corresponds to $216/6 = 36$ μsec per term on the average. This speed, very favorable in itself, is still better than it appears when grouped records are handled, because in such a case a record has to be moved to working storage area before processing. This would have required C+, and =, for each term, which with indirect addressing would have required 30.8 μsec. Thus unpacking, in effect, takes only 6 μsec per term.

It is one of the jobs of the software to relieve the programmer from the writing of the packing and unpacking routines, although these are fairly simple. For this to be possible, the programmer has to provide the software with data descriptions. These are then processed by the software to generate routines like that shown above.

In DAC this is handled by the instructions *UNPACK*, *PACK* and *MOVEITEMS*. We describe the first one briefly: *P, Disp; PR, UNPACK; P, Post; P, Area.*

"Disp" is the first of a sequence of "disposition words," one for each term in the record to be unpacked. "Post" is the first word of the packed record, and Area is first for the unpacked record. A "disposition word" indicates the bit positions of a term in relation to "Post." UNPACK processes "disposition words" in sequence, until it finds a disposition word which is zero.

D21 Problem Oriented Languages, Algol, Genious, Cobol

The position taken in designing the D21 system is that advanced problem-oriented software is to be used normally, and that with D21 hardware this does not deteriorate the processing capacity attempted.

In addition to this, experience has revealed that the language which is efficient for mathematical processing is efficient for any processing. Thus any programming language is designed to handle data and program logic in an efficient way in a computer. Thus the main features of Algol are not merely "mathematical," and those of Cobol are not merely "business-like."

We have found Algol the best language for internal computer work and Cobol Data Division a good (and the only standard) language for handling blocked and packed data. As Algol is void in this respect, we have concluded that combining Algol with an input-output

system of Cobol Data Division type is efficient and reasonably adopted to present standards. Thus a GENeral Input and OUtput System, or GENIOUS for short (Swedish: Genius), has been specified and is in the implementation stage. Genious can be used with any language but of prime interest is the combination Algol-Genious. Also, Cobol will be available for D21. In this case Genious, although partly identical with Cobol, is written in the way Algol is. Thus reserved words in Genious are indicated by underlining. Genious will do the following:

1) Control input and output automatically; *i.e.*, load input areas as soon as they are emptied and likewise unload output areas. No instructions in the program are needed for this. Terminal equipment and file assignment declarations together with *open* and *close* procedures are used to this end.
2) Prepare input records for processing and output records for output. This is done by moving and unpacking a record from an input area to a working area and the converse to an output area, as described in the preceding section. Vergs (*i.e.*, Algol procedures) like readfile (⟨filename⟩, ⟨record type⟩, ⟨end condition⟩) and writefile(⟨filename⟩) are written in the program for initiation of this activity. Record descriptions similar to those of Cobol are written in Genious part of a program and used by Genious for automatic generation of packing and unpacking sequences. We give a few of the first lines of a Genious Record Description, for illustration:

```
01 augments record type ⌜E⌝;
  02 type string size 1;
  02 new record;
    03 ne integer range 1, 10000;
    03 qty e range 1, 1000;.
```

Thus, whatever form data have on input and output media, they are prepared by Genious to be available word by word in the working area of the processing program. This then need not pack or unpack data nor move them for processing, nor do equivalent address modification. As a consequence, the processing program becomes simple, short and fast, and the absence of index registers, for instance, has little consequence. It is seen that Genious will serve data to the processing program in the form which is wanted by Algol. Thus an almost standard Algol processor can work together with Genious.

To handle data processing records Algol needs to be able to accept such statements as *if* A > B *then goto* also when A and B are strings of alphanumeric data, or to let $A := B$ be performed as the following: Move B to A also if A and B are strings or groups of data. Such an extension would be legal with Algol in its broadest sense. In order to keep the compiler simple and to comply with other Algol implementations, however, the D21-Algol-Genious system uses a set of *standard* procedures of Algol type, such as

move group	(groupname 1, groupname 2)
real to string (A, B, format)	converts the binary number A to string-form edited according to "format" and transfers it to B
string comp (a, b, c)	which compares strings a and b and stores the result as a −1, 0, or +1 according to whether a is less than, equal to or greater than b.

Two of the properties of Algol which make it so efficient as a procedures language are its facilities for handling advanced multiparameter procedures and those for handling multi-indexed (or multisubscripted) data.

Such tools are not much needed in business data processing. (The corresponding facilities are accordingly less advanced in Cobol.) In our view this is not, however, because they would not be of advantage but rather that they are not yet well understood.

Multi-indexed data call for multiplication for computing their addresses. Thus for many business computers this is very inefficient, both because their multiplication is slow and because the index registers go unused for their proper task. Multiparameter procedures need very complicated calling sequences for retrieving the parameter values, where efficient computer operations are not available for this, as we have already found to be the case for D21.

Difficulties of this kind have led computer users to abandon Cobol (although Cobol should not be blamed for them). With the design chosen for D21, such adverse arguments are eliminated; speed and memory are not difficult to use with a compiler, as are the different special circuitries. Cobol and, to a greater extent, Algol-Genious, can therefore be used well for business data processing, as well as for other jobs.

Acknowledgment

The present paper is an attempt by the writer to describe a systems design based on, and implemented by, the enthusiastic work of many people. (The responsibility for the presentation and all its weak points rests, however, entirely with the writer.)

Although all could not possible be mentioned here the author is especially indebted to V. Wentzel, chief designer of D21; B. Magnusson who provided the basis for the description of the hardware; and S. Yngvell and B. Asker for many fruitful discussions of software and systems problems.