BURROUGHS CORPORATION
Product Planning, DPS

April 15, 1963

CONFIDENTIAL

SUBJECT: Preliminary Design of a Business Data Processor

TO: J. N. Merner, Manager
Programming Systems Planning Department

FROM: D. E. Knuth, Consultant

This report describes the design of a data processor oriented towards business problems. After extensive discussions with Engineering and Product Planning personnel, I feel this design has a good deal of merit.

I feel the decisions made satisfy the "guidelines" set up for the proposed computer; the goals which were primarily stressed in this design are the following:

1. Low cost of the computer. This is by far the most important goal.

2. Simplicity of compilation and efficiency of execution for COBOL programs.

3. Emphasis on processing of magnetic tape and bulk file storage.

Of course, there are hundreds of alternatives at every step when designing a computer. With each decision I have reached, I will try to give the main reasons for adopting this policy.

Memory Unit

The memory is to be character organized, so that a single character is used or written at one time. Characters are addressed in decimal fashion, with addresses from 0 to a maximum of 99999.

The engineers say that an extremely reliable 2-microsecond memory can be built; furthermore, they seem to be unanimous in opposition to keeping parity bits in memory since they feel this is an unnecessary expense. Arguments in favor of including parity: (1) The Marketing Division feels

CONFIDENTIAL

it will be of advantage for selling computers and, (2) In my personal experience, no core memory has performed as well as the engineers claim, so only by including parity bits can we ever have proof that the memory is as reliable as they claim. Arguments against parity: (1) The memory is quite reliable and has the characteristic that when it fails, it usually does not affect only a single bit, but rather starts behaving very wildly. In such cases, the user will know enough to call the engineer promptly anyway, and the built-in test equipment on Burroughs memories enables good maintenance. (2) The extra cost of parity actually makes the machine harder to sell, conteracting the so-called sales value of parity.

An interesting fact came to light from discussion with Engineering. When "modular" memory is designed so that several sizes of memory are optional, it is relatively easy to design the memory so that the word size changes and the number of words stays fixed; it is much more expensive to have a memory which grows by increasing the number of words, leaving the word size fixed. This suprising fact can be put to good advantage; in particular, it would seem to solve the parity problem raised in the preceding paragraph, since we could easily design two sizes of memory, one with parity and one without. The customer then can buy his memory with parity at a certain extra cost if he wants; we can put a definite dollar value on parity with little extra cost to Burroughs.

The choise of memory organized by character (i.e., one word equals one character plus possible parity) was made for several reasons:

1. It causes a significant decrease in cost by decreasing the size or even eliminating registers in the processor, memory unit, and input-output devices.

2. Although a character-at-a-time memory would seem to be much slower, tests indicate an increase of running time of less than 10% for arithmetic processing, with no increase of time for input-output operations. For the highest speed input-output occurs with the bulk file which can operate at one character every three microseconds; even with two 100 KC tapes operating simultaneously, this would mean an average of a character every five microseconds, and it is possible to minimize the demand on the memory. The assumption is made here (justified below) that no two of the three operations "bulk file, tape compute" will be occurring simultaneously.

3. A machine designed to process business problems (and COBOL in particular) should preferable allow variable length fields and addressing by character. When the memory is in fact word-organized, a good deal of hardware is involved to do the automatic adjustment. Supposing the word size is five characters, a field of three characters or less can be readily accessed one character at a time, while a field of more than

three characters long has better than half a chance of being split across two or more words. On reading from memory it is a problem to assemble these words, but the problem is significantly increased when we attempt to store into memory to avoid changing the unspecified portions of the words. The simplest solutions to these problems are to do them serially with shifting registers, but this actually uses up the time which was saved by accessing several characters at once, and it means the introduction of a significant amount of circuitry and an extra "gathering" register. A fast "record transfer," a fairly common operation, is impossible on a word-organized computer unless the addresses are aligned. Tests on word length of one, five, and ten characters indicate that a greater word size causes a savings of time only in the accessing of the stored program; the remainder of the processing actually takes slightly longer to do (using the serial shifting registers) and requires much more internal circuitry. After considering a number of machine organizations, it was clear that a word-organized computer should have word-organized addressing and word-organized operation codes; therefore, it would be unsuitable for COBOL programs.

4. There is, however, a way to build the memory so that it (externally) gives the appearance of being character-organized, but the word size can be modular. The latter is important because, as mentioned earlier, a modular memory is much less expensive when the word size rather than the number of words is modular. A way to do this with a minimum of circuitry is described below; it has the effect of slowing down the memory to four microseconds in the worst case (which is rare and can be avoided when referring to disc files) and speeding up the memory to one microsecond a good part of the time. The effect of this is to allow modular memory with a much smaller price tag (in fact, getting up to three or four times as much memory at the same cost!) by letting the word size be modular; when a customer purchases additional memory, he gets a slight increase of speed as well!

The minimum memory was stated as 10,000 characters, but I believe this is unrealistic. A COBOL compiler requires extensive tables, and I believe it could be squeezed into 20,000 characters, which is perhaps equivalent to less than 3,000 words on the 220 computer, if several passes are used; but a COBOL in 10,000 characters is fairly riduculous. Therefore, I would recommend a 20K memory for the minimum configuration with expansion to 40K, 80K and 100K (by having word sizes of 1, 2, 4, 5 characters, respectively). A word size of three is incommensurate with 10 and so it is to be avoided. In this case, it would be silly to market a 60K memory if it causes the cost of all the other memories to go up significantly.

CONFIDENTIAL

If the demand for memories in all sizes from 10K to 100K, in steps of 10K, is definitely required, this can be done by allowing the capability of hooking two memory units to the computer, each of which has possible configurations of 10K, 20K, 40K, or 50K (for word size of 1, 2, 4, 5 respectively). For the scheme to be described will work without limitation, even if several different word sizes are present in memory! A 30K memory could be built up from a 20K (with word size 2) and a 10K (with word size 1). A 70K memory could be built up from a 50K and a 20K, etc. The 10,000 word memory in a 220 computer is done by having two 5,000 word memories.

Here is the scheme referred to for having the memory unit give the impression of being a character memory, while actually having larger word size. The MIR has transfer paths leading to and from all character positions. An auxiliary word register W contains the address of the word currently sitting in the MIR. The following takes place (let M be the address of the character desired and let Mw, Mc be the corresponding word address and character within the word in memory).

Action requested:	READ M.	
a.	If Mw = W,	transmit the Mc character of the MIR
b.	If Mw ≠ W,	and if previous request was a WRITE, then first do a WRITE cycle to store the MIR, then go to step c.
c.	If Mw ≠ W,	and previous request was READ, then set W = Mw and do a READ cycle. Go to step a.
Action requested:	WRITE M.	
a.	If Mw = W,	replace the Mc character of the MIR.
b.	If Mw ≠ W,	and if previous request was a WRITE, then first do a WRITE cycle to store the MIR, then go to step c.
c.	If Mw ≠ W,	and if previous request was READ, then set W = Mw and do a READ cycle. Go to step a.

Note that this READ cycle could be a destructive read-out of the core if this is advantageous. Also note that in several cases (such as reading from bulk file) the READ cycle is unnecessary here; therefore, a way to inhibit this READ cycle should be provided.

This manipulation can all be done by the memory unit, with no change required in the processor. Experiments indicate that the case Mr = W occurs frequently, especially when scanning the stored program, so that time is saved. The worst case occurs when doing simultaneous READ-READ from two tape units when the memory slows down to four microseconds per character. I have not had the opportunity to discuss with experience whether a simpler scheme (which would read, change one character, and store all in one memory cycle) is feasible; if this is true, step b. of this procedure could be omitted, and step a. of the WRITE action would be lengthened to include a memory cycle. Then the worst case would again be cut to one memory cycle (two microseconds).

To summarize the section on memory, I have presented a case for a character-at-a-time memory, as far as the processor is concerned; but a memory device can be designed with modular word size (thus, with considerably less cost than modular number of words) which gives the appearance of being a character memory.

Syllable Structure

One of the significant advances in processor design in recent years has been the advent of syllables for program strings rather than fixed format instructions, and Burroughs (with the B5000 and D825) is one of the pioneers in this field. A syllable structure provides great flexibility and decreases the memory space required for program storage since each operation can be built to use only as many syllables as it needs. The design to be described shows how a machine need not be a "one-address," "two-address," or "three-address" machine. Some operations have no addresses; some have one; some have two; some may have one hundred if the programmer desires. The machine to be described cannot only do the sequence "ADD A to B" (2-address), but it can do "ADD A and B putting the result in C" (3-address), also ADD A,B,C putting the result in D" (4-address), and many other variants, such as "ADD A,B,C,D GIVING E,F" in COBOL which can be expressed as a 6-address operation in this computer. A variable number of addresses is particularly well suited to COBOL programs, not only because of the arithmetic verbs, but because of the requirements of "multiple-precision" involved when long fields are to be manipulated.

A syllable structure also provides advantages for using index registers, allowing any number of index registers to apply to a given address. This is important for efficient handling of COBOL subscripts.

The fact that addresses may go up to 99999 would mean five BCD characters are necessary to specify the address. But even a casual glance at a typical program written for the 220 shows that a significant number of addresses are 0, 1, 2, or 9999 (meaning -1); in fact, nearly all of the index-modified addresses are of this type. A syllable structure enables these addresses to be shortened to one digit-signed addresses.

Therefore, the use of syllables tends to shorten the storage space needed for programs, tends to increase flexibility, and tends to fit more nicely with COBOL than a fixed formal instruction word. A small portion of the storage space gained by going to syllable structure is lost by the requirement that the different types of syllables must be identified from each other, but this loss is minimized by imposing certain rules of sequence on the syllables.

An interesting development comes to light when using syllables with numeric data; the least significant digit and sign are needed first. For this reason, it becomes desirable to store the successive steps of a program in descending locations rather than in ascending order. This simple change may seem rather radical at first glance; but actually, there is no reason for alarm since a COBOL user need never know what direction the program goes internally, and even an assembly program user doesn't have to make much of an adjustment. On the other hand, this change makes considerable savings in circuitry and execution time.

Let us now describe the general format for syllables before discussing detailed characteristics. Whenever we refer to the "following" or "next" syllable, we actually mean the one which will be executed next; that is, one which is stored in the preceding character position. Each syllable is a single character with positions AB8421; an attempt has been made to ensure that the syllable 111111 will never appear anywhere in a program string.

Address Syllables

Our machine has an I register (holding 5 decimal digits, no sign) which is used for indexing. Every address is normally indexed by the I register; the resetting and changing of the I register are handled by methods to be explained later. In a serial machine, it costs no extra time to have indexing by I on every address since we merely run the address through the adder rather than transfer it directly.

The address syllable one is identified by A bit = 0. If the B bit is 1, it indicates that the I register is not to be cleared after indexing. If the B bit is 0, it indicates resetting of I immediately after indexing. The 421 bits are set to 0-7 to indicate variants of the current operators as described under the operator syllable. The 8 bit is used to suppress indexing by the I register.

The address syllable two follows address syllable one. The A bit is on if the address is a literal, off otherwise; the B bit is on only if a scaling syllable follows. The 8421 bits specify the length of the operand field (except they have special meaning on certain operators); this length should

normally be from 1 to 8 for alphabetic items, 1 to 12 for numeric items, or else undefined results occur. If the address is a literal, the B8 bits of address syllable one are ignored.

The scaling syllable follows address syllable two if its B bit was on. This is used for decimal-point alignment as required by COBOL. The effect is, in general, to make the operand appear as if it were multiplied by some positive power of 10. The A bit is reserved for future specifications; the B bit is on only if rounding of the register is required (see below). The 8421 bits specify amounts of scaling desired.

If the item is used as a source field; it is effectively multiplied by 10^S and its length effectively increased by S. If the item is used as a receiving field, the register being stored is shifted S places to the right (with or without subsequent rounding) before storing the result. The scaling syllable may also be used for other purposes in certain operations.

The address itself follows, using as many syllables as digits in the address (LSD first). In the case of a literal, the length of the literal specified how many characters are given, and they are simply written in the program string (LSD first). For nonliterals, the character address is given normally to be indexed by the I register. If this address is negative, the B bit of the LSD is set on (the result after indexing should be positive). The A bit of the MSD is set on to delimit the end of the address, and all other A bits are zero.

Summary of Address Syllables

	Address	Scaling	Add Syl 2	Add Syl 1
A	end	0	literal	0
B	sign	round	scale	save I
8	d	s f	l	ignore I
4	i	c a	e	op
2	g	a c	n	vari
1	i	l t	g	ant
	t	e o	t	
		r	h	

Examples: The address of character 1002, without indexing, a three-character field, scale by 1, save I register, op-variant 1:

A		B	B
1002	1	3	9

The address "(I-register)-1," a five-character field, no scaling,

op variant 7:

Ab		B
1	5	7

The constant "1," op variant 4:

	A	
1	1	4

Addresses are usually given for LSD of the operand, except in input-output operations and other cases which are specifically mentioned. Whenever an address by MSD is given literal addresses are not allowed. This lack of consistency between MSD and LSD is done intentionally to save circuitry. An assembly program (having the field length) can do the conversion between MSD and LSD; the programmer needn't be mixed up.

Index Syllables

There are up to 16 index registers, each of which is stored in memory and consists of 5 decimal digits (no sign). The AB bits of these words are not part of the index value and may be cleared by index register operations. The memory allocation is:

Locations		Index Register Number
MSD	LSD	
00000	00004	0
00005	00009	1
00010	00014	2
.		.
.		.
.		.
00070	00074	14
00075	00079	15

Each indexing syllable simply adds the contents of the specified index register to the I register. The AB bits of an indexing syllable are set to 10 (to identify the type of syllable); the 8421 bits give the number of the index register (0-15).

Operator Syllables

The third major type of syllable is the operator syllable, identified by AB = 11. The 8421 bits give the operator class. The operator class is a number from 0-14; operator classes 1-14 are for operators with a fixed number of operands, and operator class 0 is for those operators with arbitrarily many operands.

When the operator class is zero, the following syllable gives the current operator. This "current operator" is put into a special one-character register called the C register, but no action takes place until succeeding address syllables occur. Then the variant portion of that address, together with the current C register setting, specifies the operation which takes place. The C register is not affected by operators of class 1-14; the only way it can be changed is as mentioned above.

Input-Output

Since I am by no means an authority on input-output equipment, it would be presumptuous of me to completely specify the input-output operations for the proposed system; I leave that to more capable hands. The input-output operations given here are merely suggestions showing how input-output may be incorporated into this processor design.

I have, however, spent a great deal of time considering the controversial issue of whether read-write-compute should be allowed on this system. After many discussions, I am firmly convinced that read-write-compute would raise the cost of this system out of bounds.

1. The cost of the processor is considerably increased, since a word-organized memory is necessary to account for increased load on the memory unit. This raises the cost of the memory unit also. Furthermore, a much more extensive interrupt capability would need to be built into the processor.

2. The cost of the tape control units is considerably increased, since they will need MIR and MAR registers of their own, and also command and control registers which are otherwise borrowed from the central processor.

CONFIDENTIAL

3. The increased complexity of the computer not only increases cost, but decreases reliability.

4. It is just as easy (if not easier) to write a control program which utilizes a simultaneous Read-Write (no computation) command as it is to write a control program which utilizes a simultaneous Read-Write-Compute. A Read-Write command, together with a Read-Read command, will nearly cut tape processing time in half, and the further addition of Read-Write-Compute does not actually give much of an additional saving in time.

Discussions with Engineering indicate a relatively small change in a tape control unit to work with a Read-Write command. Furthermore, a Read-Read command is very easy to add to the tape control (if there is already a Read-Write command). The extra cost of Read-Write is the cost of two tape controls rather than one, and a slight additional cost for each tape unit because some additional lines between units are necessary.

Therefore, I strongly recommend that this computer have Read-Write and Read-Read capability, but not Read-Write-Compute which involves extensive further cost. If we were planning a larger scale computer, Read-Write-Compute along with several other refinements would be preferable; indeed some more extensive multiplexing would then be highly desirable.

Registers

The computer achieves greater speed and flexibility from internal registers. Since arithmetic processing is not emphasized, arithmetic can be done serially, and the registers will be rather inexpensive since they do not need much capability (except for shifting to the right). The following registers are present.

A Register and B Register: Each has 48 bits plus a sign bit; the 48 bits hold either 8 characters (in which case, the sign has no significance) or 12 digits (BCD form). These registers are capable of shifting right. The A register can shift by 4- or 6-bit groups, but the B register can only shift by 4-bit groups.

I Register: 20 bits capable of shifting right only. This register has already been mentioned.

J, K Registers: 20 bits each; these actually are embedded in the B register. They are capable of counting up or down in steps of one. These are used as auxiliary address registers in input-output commands and the move verb.

CONFIDENTIAL

C Register: 6 bits, holds current class zero operator.

S, L Registers: 4 bits each, hold scale and length from address syllables.

Toggles 6 bits, 2 bits represent two overflow toggles (for arithmetic overflow and index register overflow) and are also used as the comparison toggle (high, low, or equal states). Four bits for tape toggles.

P Register: 20 bits, holds location of current syllable.

M Register: 20 bits, holds location of current operand. (The M register might be combined with the I register, but this would detract a little from the indexing capability as far as "save I" goes. However, some programming experience may indicate that the M register might as well be the same as the I register.)

Arithmetic Operators

The following operations we designed to implement COBOL arithmetic verbs satisfactorily, while keeping computer circuitry to a minimum. The minimum requirements of COBOL are met along with the possibility of handling long operands using multiple precision techniques.

ADD (Class 0); C register settings for Class 0 operator will be designed by Engineering.

V = 0 Load A numerically (The sign of A is set to the sign of the operand, and the A register is loaded in BCD fashion with the numeric portion of the operand. Maximum length scale factor is 12. Overflow toggle is reset.

V = 1 Store A numerically (The A register is shifted right according to the scale factor, perhaps rounded, then the numeric portion of the operand is set equal to the contents of A. The zone bits of the operand are all set to zero, except the B bit of LSD is set to the sign of A. The A register is unchanged.

V = 2 Add to A (The signed numeric operand is added into the contents of the A register. If the result is zero, the sign is set to plus. Overflow may occur.)

V = 3 On size error (The "length" field L of the operand actually expresses the maximum allowable size of the operand. If the overflow toggle is set, or if the contents of the A register is 10^L, a branch occurs to the syllable specified. The contents of A is unchanged. Note: If scaling is specified, rounding may occur as in store A; in this case, the maximum size is 10^{L+S} rather than 10^L.)

V = 6 Add to AB, upper half (Registers A,B are treated as single 24-digit registers with the sign of A. The operand is added into the upper portion. This is the same as Add to A except, if the result causes the sign of A to be changed, the B register is complemented. Overflow may occur.

V = 7 Add to AB, lower half (Analogous to V = 6, adding into B with possible carry into A, even possible overflow.)

ADD ABSOLUTE (Class 0)

V = 0 Load A absolute (Same as Load A numerically, except sign of A is unconditionally set to plus.)

V = 1 Store A absolute (Same as Store A numerically, except zone bits of the operand are unchanged.

V = 2 Add to A absolute (Same as Add to A, except operand is treated as positive.)

V = 3 On size error (Same as for ADD)

V = 4 Add to A binary (The numeric portion of the operand is added into the A register treating both as binary numbers, ignoring all signs.)

V = 5 Add absolute and store (The operand, treated positive, plus the contents of the A register which is signed replaces the value of the operand without changing any of the zone bits. The A register is unchanged. Rounding may occur, but A register is still unchanged. The result of the operation must be positive, negative results are complemented.)

V = 6 Add to AB, upper half, absolute (Same as for ADD, but operand is positive.)

V = 7 Add to AB, lower half, absolute (Same as for ADD, but operand is positive.)

SUBTRACT (Class 0)

V = 0 Load A numerically

V = 1 Store A numerically

V = 2 Subtract A <u>from</u> operand, putting result in A

CONFIDENTIAL

SUBTRACT (Class 0) - Continued

V = 3 On size error

V = 4 Add to A

V = 6 Subtract operand from AB, upper half

V = 7 Subtract operand from AB, lower half

SUBTRACT ABSOLUTE (Class 0)

V = 0 Load A absolute

V = 1 Store A absolute

V = 2 Subtract A from operand absolute (ignore zone bits of operand)

V = 3 On size error

V = 4 Add to A

V = 5 Subtract A from operand, store absolute (zone bits of operand are unchanged; A is unchanged. Rounding may occur.)

V = 6 Subtract absolute operand from AB, upper

V = 7 Subtract absolute operand from AB, lower

MULTIPLY (Class 0) Perhaps there is a better way to arrange this, but the following seems simplest for hardware and COBOL compatibility.

V = 0 Load B numerically

V = 1 Store A numerically

V = 2 Multiply register B by the operand, putting result in register A. Overflow is possible.

V = 3 On size error

DIVIDE (Class 0) More complete specifications for DIVIDE will be made later.

V = 0 Load B numerically

V = 1 Store B numerically

V = 2 Divide B into operand result in A. (The operand is first transferred to A, then the division cycles occur storing the result in some fixed reserved memory locations. Afterwords, the result is transferred to the A register.)

V = 3 On size error

V = 4 Remainder divide B into operand (Same as DIVIDE, but remainder is left in A.)

LOGIC (Class 0)

V = 0 Load A alphabetically. (6 bit characters are loaded into the A register left justified with zero filled on the right. Scale factor indicates a number of zeroes to be filled in at the left (or blanks if "rounding" specified). The length L is set to include the scale factor; L-S is the number of characters in memory. All alphabetic data movement is in this format. Maximum L is 8. The sign of register A is unchanged.)

V = 1 Store A alphabetically

V = 2 ADD (logical and with operand)

V = 3 ADD without carry (logical exclusive or)

V = 4 OR (logical or)

Data Movement

MOVE (Class 0)

V = 0 Load A alphabetically

V = 1 Move ALL (Same as MOVE unedited, except the source register is not used; the MSD of register A is moved into all characters of the destination.)

V = 2 Load sign (The B bit of the character addressed replaces the sign of register A.)

CONFIDENTIAL

MOVE (Class 0) - Continued

V = 3 Branch nonzero move (Branch to this address if overflow is on.)

V = 4 Load source (The source register J is set to the value of the address specified. The length scales are ignored.)

V = 5 Load picture (The picture register K is set to the value of the address specified.)

V = 6 Move unedited (The length of all moves is specified by the scale and length fields as 10S + L. The addresses all specify MSD rather than LSD; the desired number of characters is moved starting from the source address. Afterwards, the source address points to the character following the last character transferred. The overflow is set on if any nonzero characters have occurred in the source stream, off otherwise.)

V = 7 Move edited (Same as MOVE, but the picture register specified the MSD of an editing stream. Editing occurs as specified below.)

The following are editing rules designed to handle any COBOL construction with a transliteration of characters done by the compiler.

Character in Picture Stream	
0 or any with zone bits on	Insert this character (except on floating , is suppressed).
1 (zero suppress)	Transfer blank or zero according to zero suppress rules.
3 (check protect)	Transfer * or zero according to zero suppress rules
4 (+ sign)	Insert + if sign A is plus, - if sign A is minus
6 (- sign)	Insert blank if sign A is plus, - if sign A is minus
9 (floating $)	If first nonzero character occurs at this point, back source stream up by one and insert $ sign there.

CONFIDENTIAL

Editing Rules - Continued

	Character in Picture Stream	
	5 (floating +)	Combination of 9 and 4
	7 (floating -)	Combination of 9 and 6
Binary	12 CR	Insert two blanks if sign A is plus, CR if minus.
Binary	14 DB	Insert two blanks if sign A is plus, DB if minus.
	10 transmit alpha	Transmit source character unedited.
	11 transmit numeric	Transmit source character, set zones zero.

EXAMINE (class 0)

V = 0 Load A alphabetically

V = 1 Store B numerically

V = 2 Replace and tally all (The length of the operand addressed by MSD is given as 10S + L. For every character of this operand which equals the second character (from the left) of register A, the first character is substituted and register J (low-order part of B) is increased by one. A "rounding" specification means zone bits are ignored in the comparison and unchanged in the replacement.

V = 3 Replace and tally first

V = 4 Replace and tally leading

V = 5 Replace and tally until first

Note: If replacement is not desired, simply make the two characters in A equal to each other.

CONFIDENTIAL

Conditions and Branching Verbs

If (Class 0)

V = 0 Load A numerically

V = 1 Load A alphabetically

V = 2 Compare numerically (The contents of A treated numerically is compared with the operand treated numerically. The comparison toggles are set accordingly. The action is exactly as if the subtract operation was performed and the result tested for positive, negative, or zero; except the A register is unchanged by this comparison. Minus zero is equal to plus zero.)

V = 3 Branch on relation (The comparison toggle is set to either low, equal, or high. The value of L tells what branch is desired:

L = 0 Branch unconditionally

L = 1 Branch on low

L = 2 Branch on equal

L = 3 Branch on low or equal

L = 4 Branch on high

L = 5 Branch on unequal

L = 6 Branch on high or equal

L = 7 Branch unconditionally)

V = 4 Compare alphabetically (L left characters of A are compared with the L left characters of the operand according to collating sequence. The operand is addressed by LSD as usual. The result sets the comparison toggles to low, equal, or high.)

V = 6 Test A alphabetic (If L leftmost characters of A are alphabetic, i.e., A - Z or space, branch to the specified syllable.)

V = 7 Test A numeric (If L leftmost characters of A are numeric, i.e., 0 - 9 or + or -, branch to the specified syllable.)

PERFORM (Class 0) COBOL specifies a peculiar return mechanism for subroutines.

V = 0 Load A numerically (Used for passing parameters)

V = 1 Load A alphabetically

V = 2 Load B numerically

V = 3 Branch on relation. (See IF verb.)

V = 4 Store A numerically

V = 5 Store A alphabetically

V = 6 Store B numerically

V = 7 Record exit (The address must be "subroutine exit switch" syllable followed by a five-character address field. The syllable is set to exit, and the preceding syllables are set to the current address minus L.)

Set Index Register (Class 1)

The following syllable specifies the index register, and the B bit specifies "save I" or not. The A bit specifies "ignore I" or not, as in address syllable one. The following syllables are like numeric address syllables; they replace the specified index register. (This address may be I modified as an ordinary address.)

Increase Index Register (Class 2)

Analogous, but increasing the specified index register. The index overflow toggle is set on if overflow occurs, off otherwise.

Decrease Index Register (Class 3)

Analogous. Index overflow toggle set if result is less than zero (in complemented form) as in DBB on the 220.

Shift Right A (Class 4)

The following syllable tells in its B bit whether an alphabetic shift or numeric shift is desired. The amount of shift is given in the numeric portion. The A bit specifies whether or not the shift is circulating.

CONFIDENTIAL

Shift Right AB (Class 5)

Numeric shifts allowed only.

Clear A (Class 6)

The A register is set to plus zero.

Clear B (Class 7)

The B register is set to plus zero.

Index Branch (Class 8)

Branch if index overflow is on to the syllable specified by the next five characters.

Subroutine Exit Off (Class 12)

This is a NO-OP, skip the next five characters of the code.

Subroutine Exit On (Class 13)

This syllable is changed in memory to a "subroutine exit off" and an unconditional branch to the syllable specified by the next five characters.

No-Operation (Class 14)

This syllable is skipped.

Input-Output Operations

Tape File Processing (Class 0)

V = 1 Store B numeric (After a tape operation, the number of characters processed remains specified in the B register as the address of the character following the last one transferred.)

V = 3 Branch on tape condition (L specifies the condition and/or conditions which cause branching: 8 bit for primary error, 4 bit for primary EOT or TM, 2 bit for secondary error, 1 bit for secondary EOT or TM.)

V = 4 Load primary (Same as load source in MOVE) L = unit

V = 5 Load secondary (Same as load picture in MOVE) L = unit

V = 6 Tape I/O (S and L specify the secondary and primary operations according to the following code:

0 No operation

1 Read binary (forward) record

2 Read backward record

3 Read forward

4 Write binary (specific number of characters)

6 Write (until group mark sensed)

10 Write (specific number of characters)

V = 7 Tape control (L specifies the unit; S specifies

0 Erase tape (number of characters given by address)

1 Rewind tape

2 Backspace tape one record

3 Rewind tape with lock.)

The secondary operation may not be a Write operation. The primary and secondary addresses needed have been already loaded into the B register, as MSD (except LSD for read backward). This operation may destroy the contents of the A register. The address specifies the number of characters to be written, and it is ignored unless L = 4 or 10. If S = 0, simultaneous Read-Read or Read-Write occurs. The tape indicators are reset by this operation unless a tape error occurred at this time. A group mark is written at the end of each input record.

Auxiliary Input-Output Operations (for slower speed input-output)

V = 0 Initiate (Input from the unit specified by L)

V = 1 Accept address is ignored. (Input from the buffer of the unit specified by L)

V = 2 Select interrupt (The address is selected as the place to branch to on interrupt; this address is stored in a fixed place in memory. L specifies the unit on which interrupt is selected.)

V = 4 Branch Ready (If unit specified by L is "ready.")

V = 5 Branch Not Ready (If unit specified by L is "not ready.")

V = 6 Display (Initiate output on unit specified by L by transferring a record to its buffer. For printer, S may specify printer control.)

To minimize cost, only a very limited interrupt facility is to be built into the processor. Some interrupt processing is desirable in order to be able to do the lowest level multi-processing, as desired. This type of multi-processing would allow a card-to-tape, tape-to-printer or punch, inquiry station, or console routine to operate in conjunction with the main program. Such a requirement can be met with a very inexpensive interrupt system in which precisely one unit is selected. Whenever this unit goes from "not ready" to "ready" status, an interrupt bit is set on; this causes an interrupt to occur as soon as the current operation is finished in the processor. Interrupt means that P register, C register, and toggles are stored in a fixed place, and branching occurs to another fixed place.

Restart Processor (Class 9) Loads P, C, and toggles to restart processor after interrupt.

Bulk File (Class 0) This operation remains to be specified.

D. E. Knuth

/ss

SUMMARY OF OPERATORS

Many operators are repeated several times so that a single verb may remain in C as long as possible.

Class Zero	V=0	V=1	V=2	V=3	V=4	V=5	V=6	V=7
ADD	Load A Numeric	Store A Numeric	Add to A	On Size Error			Add to A B Upper	Add to A B Lower
ADD ABSOLUTE	Load A Absolute	Store A Absolute	Add A Absolute	On Size Error	Add A Binary	Add Absolute And Store	Add Absolute A B Upper	Add Absolute A B Lower
SUBTRACT	Load A Numeric	Store A Numeric	Subtract A From Memory	On Size Error	Add to A		Subtract From A B Upper	Subtract From A B Lower
SUB ABSOLUTE	Load A Absolute	Store A Absolute	Subtract A Absolute	On Size Error	Add to A	Subtract Absolute Store	Subtract Absolute From A B Upper	Subtract Absolute From A B Lower
MULTIPLY	Load B Numeric	Store A Numeric	Multiply By B	On Size Error				
DIVIDE	Load B Numeric	Store B Numeric	Divide By B	On Size Error	Remainder Divide By B			
LOGIC	Load A Alpha	Store A Alpha	AMD	EXOR	OR			
MOVE	Load A Alpha	Move ALK	Load Sign	Branch Nonzero Move	Load J	Load K	Move Unedited	Move Edited

EXAMINE	Load A Alpha	Store B Numeric	R and T All	R and T First	R and T Leading	R and T Until First		
IF	Load A Numeric	Load A Alpha	Compare Numeric	Branch Relation	Compare Alpha		Test A Alpha	Test A Numeric
PERFORM	Load A Numeric	Load A Alpha	Load B Numeric	Branch Relation	Store A Numeric	Store A Alpha	Store B Numeric	Record Exit
TAPE PROCESSING		Store B Numeric		Branch Tape	Load Primary	Load Secondary	Tape I/O	Tape Control
AUXILIARY I/O	Initiate Input	Accept Input	Select Interrupt		Branch Ready	Branch Not Ready	Display Output	
BULK FILE	?	?	?	?	?	?	?	?

Class Nonzero:

Class	Operator
1	Set index
2	Increase index
3	Decrease index
4	Shift right A
5	Shift right A B
6	Clear A
7	Clear B
8	Index branch
9	Restart Processor
10	
11	
12	Sub exit off
13	Sub exit on
14	No-op.