FILE MEMO June 11, 1957

SUBJECT: The Primitive Instruction Approach to Editing

By: H. C. Montgomery

Introduction

This memorandum discusses the problem of editing as treated using an approach first suggested by W. Buchholz, in May 1957. The description given here is not made with a particular machine in mind. As a result of the description, however, certain desirable characteristics for a machine which could advantageously use the scheme become obvious. Hence, in addition to introducing the new system, the discussion should also serve to aid in deciding whether the system is well suited for use in a particular machine.

Three desirable characteristics for a machine using this approach are given here. As the discussion develops the reasons for these should become apparent.

1. The data flow within the machine should be serial in nature, in terms of increments of bits called bytes. Thus, a field would be operated on one byte at a time.

2. The machine should have a set of variable field length instructions, preferably both arithmetic and utility (LOAD, STORE, etc.).

3. The machine should have at least three full word capacity registers. It is highly advisable that one of these registers have an automatic loading facility and another have an automatic storing facility. (These facilities will be described in some detail below.)

The properties which distinguish this approach are all related to the definitions of the instructions which are used, the implications they have with respect to hardware functions, and the manner in which the instructions are applied to solve editing problems.

For all of the instructions, the field upon which the operations are performed, the operand field, is understood to be the next byte or bytes in an input data register. Thus, the programmer need not explicitly identify the operand field; part of the normal execution of the instructions is to take the field from the register and operate on it. If the input data register does not contain the field upon which editing operations are to be performed, the programmer must load the word containing the field into the input data register before using the editing instructions.

The special editing instructions (called primitive instructions), can be used in two ways. The first is to use them as ordinary variable field length instructions as part of the main program. This would be the case when the operations which they are to perform would be applied to but one or two fields of input data.

The second method of application would be to use them in a manner similar to the usual way in which subroutines are used. Suppose that a sequence of operations is to be applied to a number of fields, all of the fields being similar in nature but varying in length. A subroutine is constructed which would perform these operations on the largest field which is anticipated. It is constructed, however, in a way such that shorter fields may use the same subroutine by transferring to the appropriate place in the sequence. Then each time a field upon which this sequence of operations is to be performed is encountered, a transfer is effected to the suitable place in the sequence. Examples illustrating the use of the primitive instructions in this way are given below.

The Equipment Required

The equipment required for editing using this system is shown in Figure 1. This represents only the components directly involved in editing and represents them in their minimum form. Greater execution speed could be achieved by duplicating components like the MAR to avoid delays caused by simultaneous requests for service from several sources.

It is further true that the utility register and the data path from the instruction register to the adder could be eliminated if a table look-up facility is provided elsewhere in the machine.

Finally, the description of this equipment pertains to its functions. The particular method of implementing these functions will depend on the machine involved.

The system uses three data registers, the input data register (IDR), output data register (ODR), and utility data register (UDR). All three have a length equal to that of one memory word.

The IDR can receive half words in parallel from memory, and can send serially its contents in bytes to any one of three places: the ODR, the match register, and the memory address register (MAR).

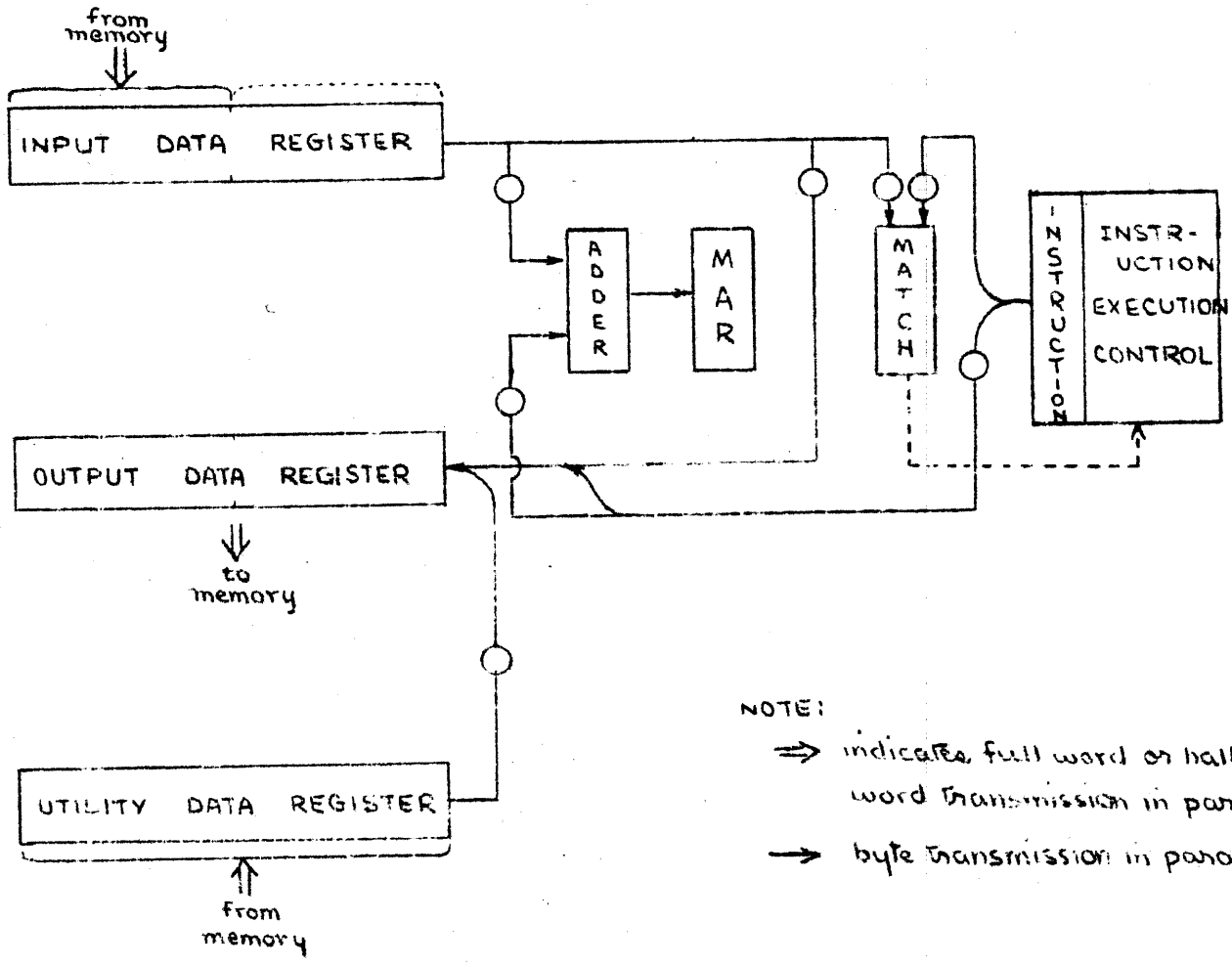
from memory
INPUT DATA REGISTER
ADDER
MAR
MATCH
INSTRUCTION
INSTR-UCTION EXECUTION CONTROL
OUTPUT DATA REGISTER
to memory
UTILITY DATA REGISTER
from memory
NOTE:
⇒ indicates full word or half word transmission in parallel
→ byte transmission in parallel

figure 1.

The ODR can recieve in serial fashion bytes from any one of three sources: the instruction itself, the UDR and the IDR, and can send its contents parallel-wise to memory.

The UDR can receive full words in parallel from memory, and can send serially its contents in bytes to but one destination, the ODR.

It is understood that the operand upon which the primitive instructions operate consists of a part of the current contents of the IDR. The programmer thinks of the operands as the next byte of input data. This next byte is identified by a bit address control and a byte size control which are associated with the IDR. The bit address gives the bit position of the left-most bit of the byte, and the byte size indicates how many bits are to be in the byte.

During the time when the machine is operating under control of the primitive instructions, the IDR and the ODR communicate with memory in a special way. As the input data are read out from left to right by bytes from the IDR, and the mid-point of the IDR is passed, the left half of the next higher-addressed memory word is automatically loaded into the left half of the IDR. Then, when the byte containing the right-most bit of the IDR is read out, the right half of the next higher addressed memory word is automatically loaded into the right half of the IDR. If, during read out from the IDR, a byte extends past the boundary of a word into the next word, the remaining bits of that byte are taken from the left end of the IDR (since the left half of the next word will have been loaded into the IDR by this time.) In analogous fashion the ODR sends its contents to memory, thus enabling the ODR to be constantly ready to receive another byte and relieving the programmer from having to concern himself with when the register becomes full and needs to have its contents stored.

The UDR need not have this automatic loading (or storing) facility for editing. Since its main role in editing is to remove certain fields from full memory words during table look-up operations, it need only be able to receive full words from memory, and send bytes to the ODR. A bit address control is used to identify the left-most bit of the field to be read out. The byte size control fixes the bits in each byte to be read out and the length of the field to be read out is given in the table look-up instruction.

A single memory address register (MAR) and adder are used for computing the correct memory address in table look-up operations. The capacity of the MAR is the same as the length of the word address field in the primitive instructions. The capacity of the serial adder is not critical.

An eight bit match register is used to identify certain input characters. When the appropriate gates are open, the next byte of input data is compared with the character given in the primitive instruction (the right-most bits of the instruction match character). A match or no match signal is sent to the instruction execution control.

Both the IDR and ODR have a word address register associated with them which records the memory address of the right half word which they contain. As mentioned above, a bit address register for each of them provides the bit address of the left-most bit of the byte being processed. When the processing of this byte is completed, the bit address will usually be advanced by the number of bits in the byte so that it will then give the address of the left-most bit of the next byte. The bit address mechanism for both the IDR and ODR is designed so that it operates on an arithmetic which is modulo the number of bit positions in the registers. Thus, when a byte begins near the right end of the register and extends past the right end of the register, advancing the bit address will cause it to count off the remaining bits from the left end of the register, giving the effect of having a circular register which is byte-wise continuous.

The instruction control box shown in Figure 1 is the same equipment as that used for controlling the execution of the standard instructions of the machine. Its function is to effect the opening and closing of the correct gates in appropriate sequence in order to perform the operations defined for the instructions being executed.

The format for the primitive instructions should as closely as possible be identical to the standard instruction format. The fields required are shown in Figure 2. Their relative positions in the word are not critical. The number of bits in each field depends upon the overall machine characteristics, such as word length, number of memory words, etc.

WORD ADDRESS	MATCH CHAR-ACTER	REPETITIONS	OPERATION CODE	UTILITY CHAR-ACTER	BYTE SIZE	I T

Figure 2

The word address field, as was pointed out above, does not give the address of the operand. It is rather used to give the address of an alternate instruction when a transfer is possible, and in the table look-up instruction to give the base address of the table.

The match character is the model with which the input bytes are compared when it is desired to detect a special character whose location in the input data is not known, as for instance, leading zeros in a numerical field.

The utility character is used as a substitute for unwanted input characters and for insertion directly into the output for such uses as punctuation.

The byte size field specifies the number of bits in each byte of input data. A second byte size field for table look-up specifies the read out byte size of the table entry. This field occupies the three right-most bit positions of the utility character field.

The repetitions field gives the number of times the instruction is to be repeated, using new input data for each repetition.

Since the format for the table look-up instruction is considerably different, it is shown here.

WORD ADDRESS	FIELD LENGTH INPUT	FIELD LENGTH OUTPUT	REPETITIONS	OPERATION CODE		BYTE SIZE OUTPUT	BYTE SIZE INPUT	I T

Figure 3

The Primitive Instructions

1. TMT (Test for Match and Transfer).

The next byte of input data is sent to the match register where it is compared with the match character sent there from the instruction. If the two bytes are identical, the utility byte from the instruction is sent to the ODR and the bit address controls of the IDR and ODR advanced. If the bytes are not identical, the bit address controls of the IDR and ODR are left unchanged and control transferred to the instruction whose address is given by the word address field of the present instruction.

Eliminated

2. TMN (Test for a Match and No Transfer).

The next input byte and the match character are compared as in the TMT instruction. If the bytes are identical, the input byte is passed unaltered to the ODR and the bit address controls of the IDR and ODR advanced by one byte. If the bytes are not identical, the bit address controls of the IDR and ODR are not advanced and control is transferred to the instruction whose address is given by the word address field of the present instruction.

Eliminated

3. TMS (Test for a Match and Skip).

The next input byte and the match character are compared as in the TMT instruction. If they are identical, the bit address control of the IDR is advanced, the bit address control of the ODR left unchanged, and the next instruction in the present sequence is taken. If they are not identical, both bit address controls of the IDR and ODR are left unchanged and control transferred to the instruction whose address is given by the word address field of the present instruction.

4. ICH (Insert Character).

The utility character is sent to the ODR a number of times equal to the number in the repetition field of the instruction. The bit address of the ODR is advanced by one byte for each insertion, while the bit address control of the IDR is left unchanged.

5. PBY (Pass Byte).

The next input byte is sent unchanged to the output area. The bit address controls of both the IDR and ODR are advanced by one byte for each repetition of this instruction.

6. BSP (Backspace).

The bit address control(for read-in) of the ODR is backed up by one byte length.

7. SBY (Skip Byte).

The bit address control of the IDR is advanced by one byte length. [A transfer is effected to the instruction whose address is given by the word address field of the present instruction.]

8. TLU (Table Look-Up).

The input field specified by the instruction , the controls of the IDR, and the contents of the IDR is added to the word address given by the instruction. The output field defined by the contents of this address and the instruction is sent to the ODR. The bit address of the IDR is advanced by the amount of the input field length and the bit address of the ODR is advanced by an amount equal to the output field length. These operations are repeated a number of times equal to the number in the repetition field of the instruction.

Note: When the repetitions of the BSP and SBY instructions cause the bit addresses of the IDR and ODR to pass through the bit positions which normally cause automatic loading or storing action, the contents of the IDR and ODR are adjusted appropriately. That is, the previous contents of the ODR are brought back and the next words for the IDR loaded in as needed.

Sample Problems

In the examples which follow, the model field will illustrate the desired output format; the primary primitive sequence (PPS) will be the instruction list which is used to begin the editing process; and the alternate primitive sequence (APS) will be the instruction list to which control is transferred when certain conditions have been satisfied. The results these programs produce for various inputs appear at the right.

The instructions in the PPS list contain in their alternate instruction address field the address of the alternate primitive instruction which appears on the same row, and therefore, which applies to the same input byte. It so happens in the examples shown here that the transfers are always from the PPS to the APS and never in the opposite direction. This is a characteristic of the chosen examples and not a general property of the system. The APS could effect a transfer to the PPS or to some other instruction list.

The examples which appear here illustrate the use of the primitive instructions in the subroutine philosophy application. The programs for applications in which the primitive instructions are included as part of the main program are obvious.

1. Suppose it is desired to edit a ten digit numerical field to put it in a dollar field format.

MODEL FIELD	PPS	APS	RESULTS Input	Output	Input	Output	Input	Output
x	TMT	PBY						
x	TMT	PBY						
x	TMT	PBY						
,	ICH(b)	ICH(,)						
x	TMT	PBY						
x	TMT	PBY	5	5	0	b	0	b
x	TMT	PBY	4	4	0	b	0	b
,	ICH(b)	ICH(,)		,		b		b
x	TMT	PBY	9	9	0	b	0	b
x	TMT	PBY	7	7	7	7	0	b
x	TMT	PBY	3	3	0	0	0	b
,	ICH(b)	ICH(,)		,		,		b
x	TMT	PBY	6	6	0	0	0	b
x	TMT	PBY	4	4	3	3	0	b
x	TMT	PBY	2	2	5	5	0	b
.	ICH(.)	ICH(.)		.		.		. b
x	TMN	PBY	8	8	1	1	0	0 b
x	TMN	PBY	1	1	2	2	0	0 b
	BSP(3)							
	ICH(b)							
	ICH(b)							
	ICH(b)							

2. The inverse problem to the preceding is to produce from an input field like $xx, xxx, xxx. xx an output field like xxxxxxxxxx. The problem here is simpler because one can assume that non-significant zeros have already been removed.

POSSIBLE INPUT	PPS	APS	INPUT	OUTPUT
($) x	SBY →	PBY		
($) x	SBY →	PBY		
($) x	SBY →	PBY		
,		SBY		
($) x	SBY →	PBY		
			$	
($) x	SBY →	PBY	x	x
($) x	SBY →	PBY	x	x
,		SBY	,	
($) x	SBY →	PBY	x	x
($) x	SBY →	PBY	x	x
($) x	SBY →	PBY	x	x
,		SBY	,	
($) x	SBY →	PBY	x	x
($) x	SBY →	PBY	x	x
($) x	SBY →	PBY	x	x
.	SBY →	SBY	.	
x		PBY	x	x
x		PBY	x	x

The gaps will not appear in the output field.

3. Another permutation of this class of problems is that which reduces fields of the form bbbxx, xxx. xx to xxxxxxx by deleting leading blanks and punctuation. The unknown quantity in this case is the number of leading blank characters. The overall length of the field is known. (This is necessary to know where in the PPS to transfer initially.)

POSSIBLE INPUT	PPS	APS	Input	Output	Input	Output	Input	Output
(b) x	TMS	PBY						
(b) x	TMS	PBY						
(b) x	TMS	PBY						
,	SBY	SBY						
(b) x	TMS	PBY						
(b) x	TMS	PBY	2	2	b		b	
(b) x	TMS	PBY	8	8	b		b	
,	SBY	SBY	,		b		b	
(b) x	TMS	PBY	7	7	b		b	
(b) x	TMS	PBY	3	3	1	1	b	
(b) x	TMS	PBY	5	5	9	9	b	
,	SBY	SBY	,		,		b	
(b) x	TMS	PBY	5	5	3	3	b	
(b) x	TMS	PBY	4	4	7	7	b	
(b) x	TMS	PBY	6	6	4	4	b	
.	SBY	SBY	.		.		b	
x	TMS	PBY	7	7	5	5	b	
x	TMS	PBY	9	9	2	2	b	

Since the length of the output field from this problem cannot be predicted, the programmer must exercise more care in its use to avoid confusing results.