Principles of Programming

Section 7: Miscellaneous Operations

IBM Personal Study Program

© 1961 by International Business Machines Corporation

Section 7: Miscellaneous Operations

7.1 Editing and Format Design

Many business applications of computers require that reports be printed. These include such things as checks and earnings statements, sales summaries, bills to customers, deduction registers, inventory summaries, etc. In the printing of most such reports, it is necessary to spend a fair amount of effort in planning for ease of readability. This area includes a number of activities such as: planning the proper spacing of the information on the report; numbering of pages; printing of headings; proper placement of total lines; insertion of dollar signs, commas and decimal points; suppression of unwanted zeros in the high-order positions of numbers. This planning of the format and appearance of reports can take a considerable amount of time, and it can also easily happen that a sizable fraction of an entire program is taken up with editing results for printing.

In this subsection we shall consider the horizontal placement of information within a line. In the next subsection, on carriage control, we shall discuss the vertical positioning of the lines on a page.

The fundamental consideration in planning the spacing of information on a line is that sufficient space must be allotted to contain the largest quantity that can ever be printed, along with at least a few additional spaces to make the reading easier. Some printing fields are of constant length; a Social Security number, for instance, always has nine digits and is almost always printed with two hyphens. Many other fields are of variable length, such as a man's name or almost any dollar amount. The first step in planning, therefore, is to determine the maximum size of each field to be printed.

Next we decide the sequence of information across the line. Sometimes this is specified in advance; other times it is left to the programmer to decide. Usually a fairly logical scheme will suggest itself. For instance, if a sales summary is to be printed, it would be uncommon to begin the line with anything but the product number. Sometimes the arrangement of information on a line—and perhaps even the spacing—is predetermined by the use to which the report will be put. For instance, W-2 forms for withholding summaries are usually available in preprinted form; the program designer must put the information in the spaces allowed. This touches on the whole broad area of forms design, which is somewhat outside the scope of this book.

Next, we consider any editing that is to be done on each field as it is printed. Most dollar amounts are printed with a decimal point, commas and a dollar sign, for instance. Naturally, these punctuation marks must be included in planning the amount of space required for each field. Very commonly leading zeros at the beginning of the number are deleted in printing.

The computer techniques by which the fields are printed with punctuation marks and by which zeros are suppressed depend naturally on the instructions available in the particular computer. As we have seen, there are several specialized instructions available in the 1401 that greatly simplify editing. The Move Characters and Suppress Zeros instruction makes a simple matter of deleting the leading zeros, if that is all that is required. The Move Characters and Edit instruction, as we have seen, greatly simplifies the insertion of punctuation symbols and, in fact, is able to do a good deal more. It may be worth while to review the basic functions of this instruction before pointing out one or two additional features of it.

The Move Characters and Edit instruction requires that an edit control word be placed in the output storage area before the data is edited. This edit word will contain any punctuation marks that are to be inserted in the field. When the instruction is executed, characters from the A-field are moved to the B-field, working from right to left, except that characters in the B-field other than zero and blank are not disturbed. This means ordinarily that dollar signs, commas and decimal points are left unchanged in the B-field. However, almost any other character may be put into the edit word and will also be left unchanged. An exception is the ampersand, which, if present in the edit word, will be replaced by a blank and the blank not disturbed by the movement of characters from the A-field. This makes it possible to insert blank spaces in the edited field—for example, between a dollar amount and the letters CR (for "credit").

If suppression of leading zeros is desired, a zero should be inserted in the control word. After the A-field has been transferred to the B-field, leading zeros will be deleted down to and including the original position of the zero. The point of this qualification on the position of the zero is that there is usually a limit to the amount of zero suppression desired. For instance, if we have set up the program to print amounts in dollars and cents, we ordinarily want amounts under one dollar to be printed in the form .XX. The limit is indicated to the machine by the position of the rightmost zero in the control word. The zero suppression part of the editing operation also replaces with blanks any commas to the left of the first significant digit in the field.

The editing operation can be used to do something else. In planning output formats it is always necessary to decide what is to be done with negative amounts. These are most commonly printed with a minus sign;

they can also be indicated by the letters CR. The question arises: If the field to be printed can be either positive or negative, how do we handle the decision as to whether or not to print the minus sign (or the credit symbol)? The Move Characters and Edit instruction makes provision for this problem. To describe its action we must define the *body* of the control word and the *status* portion of the control word. The body is the part beginning with the rightmost blank or zero and continuing leftward to the character at which the A-field word mark is sensed. The remaining portion of the control field is referred to as the status portion. The handling of negative amounts is as follows: We insert the minus sign or the characters CR, whichever is desired, in the status portion of the control word. The Edit instruction automatically determines whether the number in the A-field is positive or negative. If it is negative the minus sign (or the CR) is left in the field; if the data field is positive, the symbols are blanked out.

For an example of the use of these features, consider the printing of a line of an accounts receivable register. Shown below are the information that must be printed, the symbol assigned to each field in the program of Figure 1, and the maximum number of characters in each field.

Field	Symbol	Maximum Number of Characters
Customer Number	CUSTNO	5
Customer Name	CUSTNA	25
Invoice Number	INVNO	5
State	STATE	2
District	DIST	2
Invoice Month	MONTH	2
Invoice Day	DAY	2
Invoice Amount	AMOUNT	6

The customer name is alphabetic and is to be printed exactly as it appears in storage. All the other fields except the dollar amount are to be printed with simple zero suppression. The date is to be printed in two columns to separate the month and the day. The invoice amount is to be printed with the dollar sign, comma and decimal point. It must also be printed with a CR if the amount is negative. (This would, of course, indicate an "invoice" sent out to show a credit from an overpayment.) The CR is to be printed one position to the right of the amount—that is, with one blank between the pennies and the C.

In the absence of a predetermined format or an existing form that *must* be used, we are free to make our own decision as to the order in which these data fields should be printed on the line, as well as their

spacing. It seems reasonable to begin the line with either the customer number or the customer name, and to group the customer name, number and location at the left of the line. The invoice number, date and amount could reasonably be printed in that order toward the right side of the line. Let us agree rather arbitrarily to print the customer name first. Now, considering the editing symbols that will be inserted in the invoice amount, and allowing, say, three spaces between fields, and assuming that we begin printing with print position 1, we arrive at the following assignments for the fields on the report shown below:

Field	Position
Customer Name	1-25
Customer Number	29-33
State	37-38
District	42-43
Invoice Number	47-51
Month	55-56
Day	58-59
Amount	63-74

The program segment required to set up the printing line once the data is in the symbolic location shown is presented in Figure 1. For simplicity in studying the principles of editing, the addresses are shown in absolute. We understand that in ordinary practice these should be symbolic.

Notice that the edit control word is shown with an ampersand, which will cause the edited field to contain a blank space at that position. The credit symbol is shown as the characters CR; these will be deleted by the execution of the instruction if the amount is positive (as it, of course, will be in most cases). The zero in the edit control word indicates the rightmost limit of zero suppression. The comma will be deleted if the amount is less than a thousand dollars. If zero suppression does occur, there will be blanks between the dollar sign and the first digit of the amount. (An optional feature on the 1401 called expanded print edit would make it possible to move this dollar sign so that it would be immediately to the left of the first significant digit.)

Review Questions

1. *What does the zero in an edit control word do?*
2. *Name some of the considerations in deciding on the placement of information in a printed line.*

IBM **1401** Symbolic Programming System
Coding Sheet

Program ____________ Programmed by ____________ Date ____________

Page No. 1 of 1 Identification ____

LINE	COUNT	LABEL	OPERATION	(A) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	(B) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	d	COMMENTS
010			MCW	CUSTNA				0225					
020			MCS	CUSTNØ				0233					
030			MCS	STATE				0238					
040			MCS	DIST				0243					
050			MCS	INVNØ				0251					
060			MCS	MØNTH				0256					
070			MCS	DAY				0259					
080			LCA	EDITWD				0274					
090			MCE	AMØUNT				0274					
100			H	*	-	003							
110	12	EDITWD	DCW	*		$,		0.	&	CR			
120													
130													
140													
150													
160													
170													
180													
190													
200													

Figure 1. Program illustrating editing operations.

7.2 Printer Carriage Control

Control of the vertical spacings of lines on a report is necessary for a variety of reasons. Sometimes a heading line must be printed at the top of a page and separated from the body lines by one or two lines of space. Often a preprinted form requires that the printing appear in specified positions on the form, making it necessary to space the paper to these positions before printing. Sometimes there is a variable amount of information to be printed on each page, followed perhaps by a total line, after which the form must be spaced to the top of the next page. This might happen, for instance, if all the purchases by one customer have to be listed starting on a new page, followed by a total line. After this, of course, the information for the next customer should start at the top of a new page.

Control of the spacing of the output document is accomplished by a combination of programmed signals to the printer carriage and a control tape in the carriage itself. The control tape is prepared for each application, or each group of similar applications, with holes in proper positions to indicate where carriage spacing is to stop once it is started. The tape-reading mechanism is shown in Figures 2 and 3. The mechanism is seen to be in some ways analogous to a card-reading system. That is, brushes are kept from making contact with an electrically charged roller except where holes appear in the loop of paper tape.

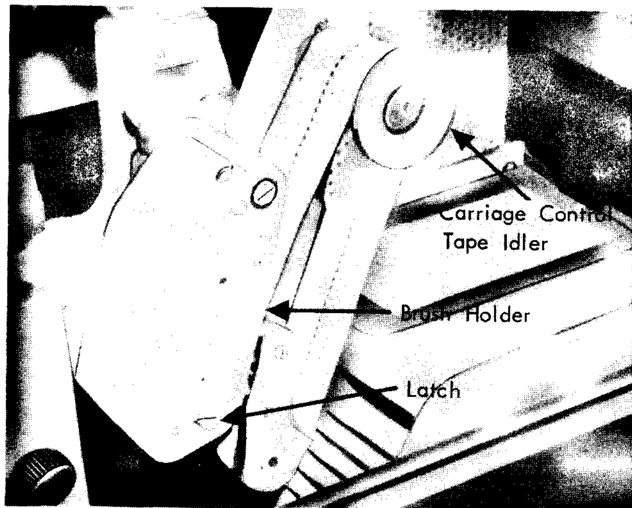

Figure 2. Tape-reading mechanism of the tape-controlled carriage in the 1403 Printer.

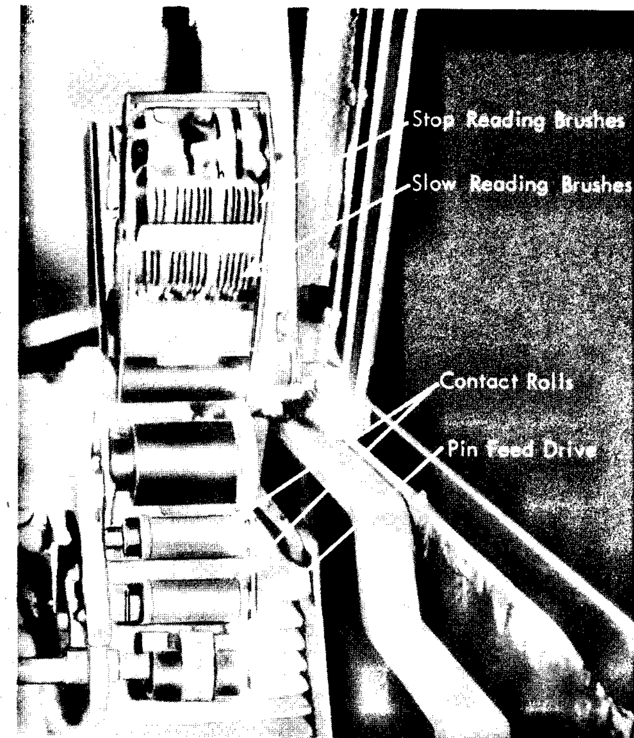

Figure 3. Carriage tape brushes.

A control tape has twelve columns of positions indicated by vertical lines. These positions are called channels. Holes can be punched in each channel throughout the length of the tape, which is ordinarily the same length as one complete page. A maximum of 132 lines can be used to control a form, although for convenience the tape blanks are slightly longer. Horizontal lines are spaced six to the inch, the entire length of the tape, which corresponds to the height of one line of printing. If the form has fewer than 132 lines (most do), then the tape can be cut off at the desired length. Round holes in the center of the tape are provided for the pin feed drive that advances the tape in synchronism with the movement of a printed form through the carriage. The effect is exactly the same as if the control holes were punched along the edge of each form.

At any point in a program where it is desired to cause skipping of the form to a new printing position, a signal can be given using the Control Carriage instruction. The d-character of this instruction specifies which channel of the tape is to stop skipping, as shown in the summary box. For example, if a skip to channel 6 is called for, the paper will start moving and will stop only when a hole in channel 6 is detected. Depending on the purpose of the form control operation, there

may be only one hole in a channel or several holes, and, of course, there may be unused channels. The Control Carriage instruction may also be used to cause the spacing of one, two or three lines not under control of the control tape. This function is also shown in the summary below:

Control Carriage

FORMAT	Mnemonic	Op Code	d-character
	CC	F	x

FUNCTION This instruction causes the carriage to move, as specified by the d-character. A digit causes an immediate skip to a specified channel in the carriage tape. An alphabetic character with a 12 zone causes a skip to a specified channel after the next line is printed. An alphabetic character with an 11 zone causes an immediate space. A zero-zone character causes a space after the next line is printed. The table shows the function of the d-character. If the carriage is in motion when a Control Carriage instruction is given, the program stops until the carriage comes to rest. At this point, the new carriage action is initiated, and then the program advances to the next instruction in storage.

d	Immediate skip to	d	Skip after print to
1	Channel 1	A	Channel 1
2	Channel 2	B	Channel 2
3	Channel 3	C	Channel 3
4	Channel 4	D	Channel 4
5	Channel 5	E	Channel 5
6	Channel 6	F	Channel 6
7	Channel 7	G	Channel 7
8	Channel 8	H	Channel 8
9	Channel 9	I	Channel 9
0	Channel 10	?	Channel 10
#	Channel 11	.	Channel 11
@	Channel 12	⌑	Channel 12

d	Immediate space	d	After print-space
J	1 space	/	1 space
K	2 spaces	S	2 spaces
L	3 spaces	T	3 spaces

WORD MARKS Not affected.

TIMING $T = .0115\ (L_I + 1)$ ms plus remaining form-movement time, if carriage is moving when this instruction is given. The form-movement time is determined by the number of spaces the form moves. Allow 20 ms for the first space, plus 5 ms for each additional space.

It is essential in using the two types of spacing (not skipping) to realize that there is normally one space after printing. If an *immediate* space is used, there will be as many lines spaced over as are called for by the d-character. If a J is written, one line will be spaced, etc. After spacing, the line is printed and then the paper is spaced one line as normally. When spacing *after* printing is called for by writing a slash, S or T, the number of spaces prescribed will be the *total* number of spaces after printing *including* the one that normally occurs. Thus, a Control Carriage instruction with a d-character of slash has no net effect. A d-character of S will call for one *additional* space and a d-character of T will call for two additional spaces. Finally, it must be realized that whereas the immediate skip and the immediate space cause the requested action to take place as a result of *this* instruction, the skip after printing and the space after printing become effective only *after* the next line is printed. An after-print skip or space would have no effect if another line were never printed.

If the carriage is already in motion when a Control Carriage instruction is accessed, the program waits until the carriage comes to rest. At this time the new carriage action is initiated and then the program advances to the next sequential instruction.

The Control Carriage instruction as we have described it is a two-character instruction. It is also possible, however, to write an I-address, in which case the instruction is called Control Carriage and Branch. After carrying out the prescribed carriage action, the next instruction is taken from the location specified by the I-address.

One of the commonest and at the same time simplest forms of operation is skipping to a new page when one page has been printed, perhaps after first printing a total line. There are two rather different ways to sense the end of a page. One way is to set up a program counter to count the number of lines already printed. When this counter reaches the number of lines in a complete page of the particular report, a Control Carriage instruction can be executed. (Although there is no logical necessity for doing so, the skipping to the top of a new page is most commonly controlled by a punch in channel 1.)

The other way is to put a punch in channel 9 or 12 in a position corresponding to the last printing position on the page. The detection of a hole in either of these channels turns on a corresponding indicator,

which stays on until a hole in another channel is sensed. This makes it possible to print lines without counting them and detect the end of the page by detecting the proper punch in the carriage control tape. This has the advantage of not requiring a program counter, which can sometimes be inconvenient.

Whichever method of end-of-page detection is used, we often set up the signal so that it indicates only the end of printing in the *body* of the page. A typical page format consists of a heading line, a certain maximum number of body lines and a total or summary line. The signal that the end of the page is about to be reached is needed when the last body line has been printed. We then commonly skip a line before printing the total, and then go on to the next page. The presence of two additional lines after the last body line must, of course, be taken into account in setting up the constant against which the line counter is tested or in punching the hole in channel 9 or 12 of the carriage control tape.

Review Questions

1. State precisely what action is caused by the instruction
CC 0800 S
2. What would you do if a form were so short that the corresponding length of carriage control tape was not long enough to go around the tape-reading mechanism?

7.3 Input and Output Timing

The discussion so far has said little about the timing of reading or punching a card or printing a line. The maximum speeds have been given, but these are hardly the whole story—in the 1401 or in any other computer. We must be concerned also with a number of other questions:

1. If the maximum speed cannot be obtained, does the speed drop to some lower figure in a large jump?

2. For what portion of the total reading or writing cycle is the computer waiting on the input-output operation and unable to do processing? Conversely, for what portions of the total cycle is it possible for the computer to be carrying out processing?

3. At what point during the cycle for one operation is it necessary to give the impulse to start another one if the device is to operate at maximum speed?

Questions of this general sort must be considered in planning the programming of input and output operations in any computer. However, the features of individual machines vary so greatly that it is hard

to make generalizations about all machines. Therefore, we turn to a detailed consideration of the 1401 as generally indicative, although not everything we shall say applies exactly in the same form to other machines.

Card reading in the IBM 1402 is carried out at a maximum speed of 800 cards per minute. This works out to 75 ms for the reading of one card. The 75 ms are divided into three portions, as shown in Figure 4. The *read start time* of 21 ms is the interval between the starting of the cycle and the time when information actually begins to move into core storage. It is spent in moving the card from the hopper to the point where the 9 row is under the brushes and information starts to be transferred into storage. The card-reading time of 44 ms is taken up with the reading of the twelve rows on the card and the transfer of the information into storage. The remaining 10 ms of processing time may be used for processing the information on this card, and still maintain the reading of cards at maximum speed. If the instruction to read the next card is executed before the 10 ms processing time is completed, cards will read at full 800-per-minute speed; if the processing time exceeds 10 ms, then the card-reading speed drops in one single jump to 400 cards per minute.

This jump is caused by the fact that there is only one point in the cycle of the card reader at which an impulse to start the card-reading operation can be obeyed.* If the impulse comes before the end of processing time, it will be obeyed—that is, another card will be read, without any delay. If the impulse comes after the end of processing time, the mechanism will wait for another complete cycle to elapse before obeying the impulse. This means that there is no steady card-reading speed between 800 per minute and 400 per minute. It can happen, however, in some cases that the following card will be read with no delay and in other cases that there will be a delay of one or more cycles between the reading of successive cards. In such a case the *average* card-reading speed may be some intermediate figure.

It is important to realize that the computer is completely idle during read start time and card-reading time. Stated otherwise: When a Read a Card instruction is executed, the next instruction is not executed until card-reading time for that instruction has been completed—that is, a minimum of 65 ms later. We say that the computer is *interlocked* during the read start time and the card-reading time. (A special feature called *read release* is available for the 1401 to make the read start time available for processing.)

*An optional feature called Early Card Read provides three starting points, thereby speeding up card reading considerably in some applications.

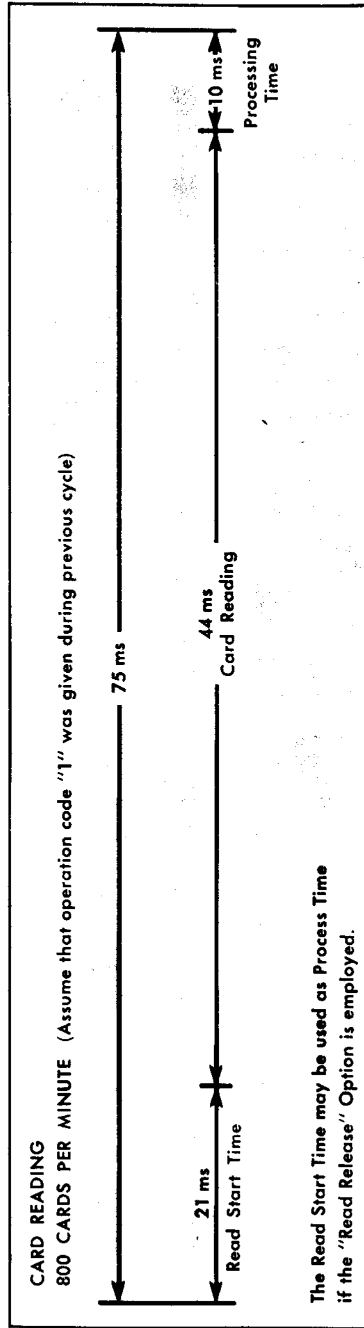

Figure 4. Timing diagram for card reading with the 1402 Card Read Punch.

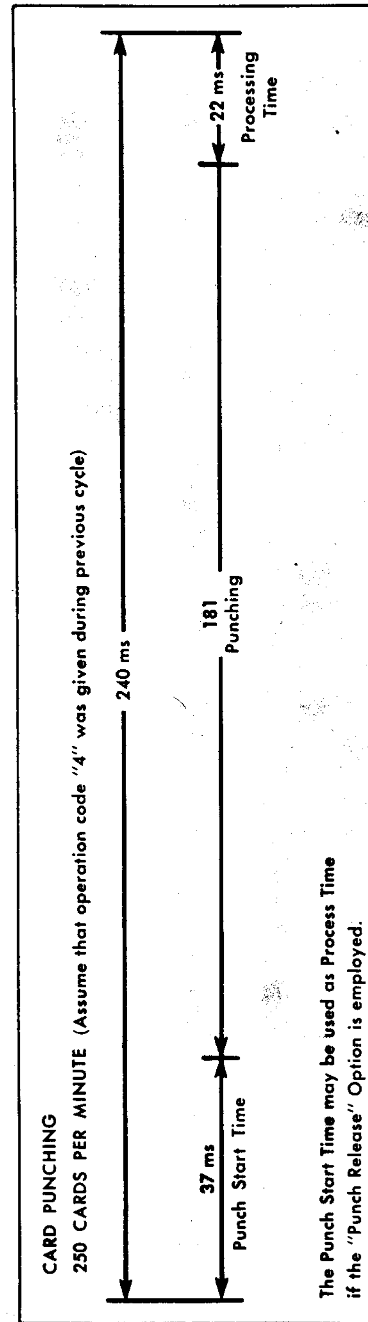

Figure 5. Timing diagram for card punching with the 1402 Card Read Punch.

Card punching is carried out at a maximum rate of 250 cards per minute, which works out to 240 ms per card. This cycle is divided into three parts also, as shown in Figure 5. The punch start time of 37 ms is the interval between the start of the card motion and the beginning of actual punching. The punching time of 181 ms begins with the 12 row. The 22 ms remaining is available for processing.

The computer is interlocked during punch start time and punching. (If a special feature called *punch release* is installed, punch start time is available for processing.) To maintain full card-punching speed of 250 per minute, the instruction to punch the following card must be executed before the end of processing time. However, in the case of punching, there are four points during the cycle, occurring at 60-ms intervals, at which an impulse to start punching can be obeyed. Therefore, if the instruction to punch another card is given shortly after the end of the part of the cycle shown as processing time, the punching speed will not be slowed down to half of the maximum. Instead, the following card will take 300 ms—that is, the normal 240 ms plus the 60 ms during which the computer will wait until another punch impulse can be accepted. Thus the "penalty" for not getting the instruction to punch another card executed during processing time, is not as heavy with punching as it is with reading.

Printing is carried out at a maximum of 600 lines per minute, which is 100 ms per line. The cycle is divided into two basic parts, with another part of the total operation overlapping one of these, as shown in Figure 6. The printing time is 84 ms; during this period the computer is interlocked. The remaining 16 ms are available for processing; if the next print instruction can be given during this processing time, printing will be carried out at full speed. As we have seen before, printing is always followed by a single line space, unless a Control Carriage instruction has been executed to specify otherwise. The normal single spacing takes 20 ms, which completely overlaps processing time. It is important to note that any skipping which may have been specified, either immediate or after-print, does *not* overlap any of the processing time. On the other hand, the computer is not interlocked during the skips unless the skip instruction happens to be executed when the carriage is already in motion, in which case the computer is interlocked only until the previous movement is completed.

The printer is able to accept an impulse to print a line at any time. If the instruction to print the next line can be given before the end of processing, printing will proceed at full speed. If the instruction to print the next line cannot be given during processing time, the only time penalty is the excess over processing time; we do not have to wait until some specified point in the following cycle. In short, the printing cycle begins whenever the Write a Line instruction is executed.

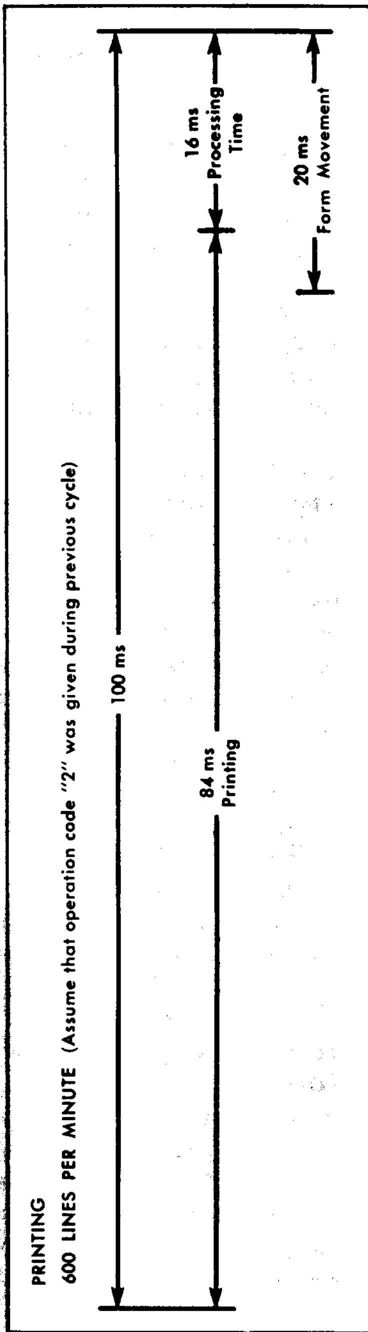

Figure 6. Timing diagram for printing with the 1403 Printer.

The total time for input and output operations in the 1401 can be somewhat reduced if it is feasible to use combined operations. The Write and Read instruction, for instance, combines the functions of Read a Card and Write a Line. Its mnemonic operation code is WR and its actual operation code is 3. When this instruction is executed, the printer takes priority and the print cycle is completed before the actual card-reading operation takes place. However, the execution of the instruction is set up so that the signal to start the reader is accepted before the end of the print cycle. Thus, read start time overlaps the print cycle, with a reduction in the total time required for the two operations. This total time is 150 ms, of which the last 21 ms are available for processing.

The Read and Punch instruction (mnemonic RP and actual 5) combines these two operations with an even more favorable overlap. Here, the two operations occur simultaneously, with the total time being that for punching a card, 240 ms. Time available for processing is 22 ms.

Write and Punch (mnemonic WP and actual 6) combines these two functions. The situation on overlaps is about the same as with Write and Read: the printer takes priority but the start punch signal is automatically given by the machine before the end of the print operation. Therefore, the punching begins very shortly after the printing is completed. The total time for the two output operations is 300 ms, of which the last 28 ms are available for processing.

The Write, Read and Punch instruction combines all three operations. The mnemonic code is WRP and the actual code is 7. Here the printing takes place first, immediately after which reading and punching occur simultaneously. The total time is 300 ms, with the last 28 ms available for processing.

The effective use of these combined instructions naturally depends on rather careful planning of the program, to insure that the information in the read, punch and print areas is set up in such a way that the combined operations produce correct results. To take a simple example, suppose that we are required merely to read a deck of cards and print the information on each of them. We immediately think of using a Read and Write combination. However, this can obviously not be done on the *first* card, because as we begin there is nothing in the print area to be printed. Therefore, we start with a Read a Card instruction. After the information has been moved from the read area to the print area, possibly with rearrangement of the data fields and editing, the Read and Write combination can effectively be used to print the information in the print area and then immediately read another card without having to wait for the read start time. In every case after setting up the information in the print area we make a last-card test and, when it is satisfied, execute only the Write instruction.

Review Questions

1. *Outline the timing differences between Read, Punch and Print.*
2. *Under what conditions can a complete job be done in just the time required for input and output?*
3. *How much time does the WRP instruction save over doing the three operations separately?*

7.4 Buffering

In most business data processing applications there is a relatively large amount of input and output. The total time required to do the job is often largely taken up with reading and punching cards, printing reports, and, as we shall see in the next section, reading and writing tapes. We have seen in the preceding subsection that in the case of the 1401 a relatively small amount of the input and output time is available for processing. In the absence of the buffering facilities to be discussed now, this is true in most computers. In fact, in some machines the *entire* cycle is unavailable for processing. This can very well mean that the total time to do the job is the sum of the times required for each of the individual input and output operations plus the total processing time. If the processing time is very much less than the input and output time, or if the processing takes a great deal longer than the input and output, there is really not much to be saved by trying to overlap the input and output with processing.

However, it frequently happens that the input and output time is about the same as the processing time. In such a case, it becomes very desirable to set up the machine so that processing can continue during most of the time required for the input and output cycles. To take a specific instance, consider printing. We would like to be able to move the information to be printed from core storage to a small auxiliary storage (this can be done at electronic speeds) and then continue with normal processing while the information is moved from the small storage to the printer at the mechanical speeds of the printing device.

The following is the essence of *buffering*: On output, information goes from core storage to the small auxiliary storage, which is called the buffer. Since no mechanical operations are required for this transfer between two electronic devices, it can be done at very high speeds. Then processing may continue while the information is sent out to the output device at the speeds required of the mechanical device. On input, the process is simply reversed. Information is accumulated in the buffer storage as the input medium is read and, when all the information has been assembled, is transferred to core storage at high speed.

Some computers have no buffering; others buffer virtually every input and output operation. The 1401 can be equipped with an optional special feature called *print storage*, which puts it in an intermediate class. Print storage gives us buffering of printing only. However, since in many applications printing time is a fairly sizable fraction of the total job time, this can mean a very significant reduction in the time required to do each job and, therefore, an increase in the data processing capability of the equipment. This is all the more true because print storage permits virtually all of the print cycle to be used for other processing, including other input and output operations. Thus, for instance, it is possible to keep the printer and reader both running at 400 per minute and still have over half of the total job time available for processing.

The 1401 print storage feature operates in just the manner described for output buffering in general. When the Write a Line instruction is executed, the information in the print area, positions 201 through 300 (or 201 through 332) is moved to a special nonaddressable buffer storage. This transfer requires only 2 ms, and it is only during these 2 ms that the computer is interlocked from processing. As soon as the information has been moved to the print storage buffer, processing can continue for the remaining 98 ms of the print cycle. As we have noted, it is possible to initiate other input-output operations during this time. If another Write a Line instruction is given before the completion of the total 100-ms print cycle, the computer will interlock until the completion of the previous cycle, at which time the next cycle will begin immediately.

The effective use of buffering requires a certain amount of preplanning of the program organization. For instance, if two Write instructions are given in sequence, then the execution of the second one will be interlocked until the first one is completed. Buffering will have saved nothing on the first instruction. Therefore, whenever possible, we try to space out the printing operation so that a computer will be interlocked as little as possible.

In the 1401, with its capability of buffering only one operation, the planning requirements are not really severe. Even if the programmer gives no special thought to buffering, and simply puts his Write instructions wherever he would put them if the machine did not have print storage, the feature will save a certain amount of time although it may not be used to full advantage. In some of the larger computers, however, where all input and output operations are buffered, the effective use of the complete computing system requires very extensive programming systems to attempt to keep all of the components of the system in operation as much of the time as possible. These input and output packages are prepared by a special programming group and are then utilized by all other programmers.

Review Questions

1. *Buffering can be described as a way of matching the speed of electronic storage with the much slower speeds of input and output devices. How does buffering save time?*
2. *Why does buffering not save much of the total percentage of job time when processing already takes much longer than input and output?*

7.5 Program Timing

It is frequently necessary to estimate the amount of time that a program will require. Obtaining an accurate estimate of this sort requires a number of pieces of information and careful consideration of a variety of factors that affect the total time to execute the program.

The basic idea is simply to take the total time required for input and output, add to this the total time required for the execution of internal processing instructions, and subtract off the amount of any overlapping of input and output with processing. Doing this requires, first of all, estimates of the total amount of input and output, together with timing information on these operations. This much is fairly simple, provided that the volume estimates are reasonably accurate.

The timing of the internal processing operations is a little more difficult. The time required to execute each individual instruction is fairly readily obtained from the programming manuals. This information has been shown in the instruction summaries throughout this manual, and will be described shortly. This, however, is not the end of the estimating job. Complexity is added by the fact that most programs have alternative paths that may be followed for different conditions existing in the input. Often these paths are not of the same length, so that the total time must be derived from a weighted average based on the expected fraction of the time that each of the paths will be followed. And this must be a *weighted* average. If the normal path for processing information on a card takes 40 ms, whereas in special cases, arising only two percent of the time, the processing takes only 10 ms, then one gets a very misleading picture of the total time if the average time for processing one card is taken to be 25 ms. This is one source of complexity.

A second and more serious complication is the fact that processing is often *partially* overlapped with input and output. Related to this problem are any considerations such as the fact that in the 1401 the card-reading cycle can begin only at specified times. It can also very easily happen that the processing for some types of data will be *completely* overlapped with input-output, whereas the processing of other

data that takes longer will be only *partially* overlapped. This can lead to erroneous estimates if the "variable overlapping" is not taken into account.

For instance, an average processing time may be short enough to allow complete overlapping, but this sort of "average" is very misleading. The time "lost" on the longer-than-average processing cases is not offset by the shorter-than-average cases, because once the processing is completely overlapped there is no more time to save.

This is not the place to enter into a complete and detailed explanation of how to handle all these considerations, since the subject depends too strongly on the features of the particular computer being used. We shall have to be content with the observation that if high accuracy of time estimates is required, then extreme care must be exercised in making the time estimate. Carelessly made time estimates are notoriously inaccurate.

We may close this very brief consideration of time estimating by mentioning the 1401 timing formulas given in the summary boxes for the various instructions.

The timing of the IBM 1401 is described in terms of the time required for one complete core storage cycle, which is 11.5 μs (microseconds, or millionths of a second) or 0.0115 ms (milliseconds, or thousandths of a second). The time required for any internal processing instruction is always a multiple of this interval of time. The timing formulas are given in terms of certain characteristics of the instruction under consideration and of the data fields being operated upon. The symbols used for these variables are shown in Figure 7.

SYSTEM TIMINGS

Key to abbreviations used in formulas

Symbol		Meaning
L_A	=	Length of the A-field
L_B	=	Length of the B-field
L_C	=	Length of Multiplicand field
L_I	=	Length of Instruction
L_M	=	Length of Multiplier field
L_Q	=	Length of Quotient field
L_R	=	Length of Divisor field
L_S	=	Number of significant digits in Divisor (Excludes high-order 0's and blanks)
L_W	=	Length of A- or B-field, whichever is shorter
L_X	=	Number of characters to be cleared
L_Y	=	Number of characters back to right-most "0" in control field
L_Z	=	Number of 0's inserted in a field
I/O	=	Timing for Input or Output cycle
F_m	=	Forms movement times. Allow 20 ms for first space, plus 5 ms for each additional space
T_m	=	Tape movement times
Σ	=	Number of fields included in an operation

Figure 7. Abbreviations used in instruction timing formulas.

For an example of how these formulas are applied, consider the equation for the Move Characters to A or B Word Mark, which is:

$$T = .0115\,(L_I + 1 + 2\,L_W)\,\text{ms}$$

Looking at Figure 7, we see that L_I stands for the length of the instruction and L_W stands for the length of whichever data field is shorter. A Move instruction without chaining has seven characters. Suppose that we are moving a field of 11 characters. The total time is therefore:

$$.0115\,(7 + 1 + 2 \cdot 11)\,\text{ms} = 0.345\,\text{ms}$$

We may note in passing how these storage cycles are used. It clearly takes one cycle to get each instruction character from storage to the control registers, and one extra to get the operation code of the next instruction and recognize its word mark. This is the basis of the $L_I + 1$ in the formula.

The movement of each character of the data field takes two storage cycles: one to get it from the A-field and one to place it in the B-field. Thus the number of cycles spent in data movement is twice the number of characters moved, which in turn is the number of characters in the shorter field, in the MCW instruction.

The formulas for most of the instructions are equally simple. A few of the more complex instructions have correspondingly complex formulas. Multiplication, for instance, is a fairly long instruction and, furthermore, depends not only on the length of the field but also on what the digits in the field are. The formula that is shown is fairly complex and, at that, gives only an average. However, in most cases the computation of the time required for the instruction is not at all difficult.

It will be noted that the input and output instructions show the time in two parts. One part gives the time required in the central computer, to which must be added the time taken up in the actual input or output actions. The time for these latter cannot be given as fixed numbers because of such variables as the restricted number of points at which a card-reading or card-punching cycle can begin and the fact that the *effective* time of these operations depends on whether the processing time can be used for processing or whether the program is organized so that the computer will be interlocked during part of the processing time. Therefore, as we have noted, the estimation of input and output operations depends not only on the way the computer is built but also very strongly on the way the program is organized.

7.6 Subroutines and Utility Programs

A subroutine is a group of instructions that performs some well defined segment of a data processing operation. Subroutines are of two broad types and are used for two rather different reasons.

One frequent reason for using a subroutine is that someone else has already written it and it can, therefore, be incorporated in a program with little effort. For instance, in the basic 1401 there is no multiply instruction. It is not unduly difficult to program multiplication, but it takes more effort than a programmer wants to have to expend every time he needs to multiply. Fortunately, there is no need for him to do so: routines are already available for the purpose. All that the programmer has to do is to obtain the cards on which the subroutine is punched and insert them in his program deck before assembly. Knowing that after assembly his object program will contain the multiply subroutine, he can write instructions in his program to use the subroutine without having to spend any time in programming the multiplication.

If multiplication is required only once or twice in a program, it is satisfactory to insert the subroutine right where it is needed in the program. The only extra operations required are those necessary to place the multiplier and multiplicand in standard locations where the subroutine can find them, and to retrieve the product from a standard location where the subroutine puts it. Since the subroutine falls right in the sequence of instruction execution, there is obviously no need to branch to it. This is the essence of the *open subroutine*—that is, it is inserted in the main program where it is needed, and appears in the program as many times as it is needed.

Suppose, on the other hand, that multiplication is required at a dozen different places in a program. Now we begin to worry about the storage space that is wasted by having the same subroutine at a dozen places in storage. Why not put it in just once, and then branch to it whenever a multiplication must be performed? Now we have a *closed subroutine*. To summarize, a closed subroutine is placed in storage at *one* place; whenever the subroutine is needed, the main program branches to it, and the subroutine branches back to the main program when it is finished.

This does create one new problem: How does the subroutine know where to return when it is finished? This question is answered by a *linkage*. Before branching to the subroutine, one or two instructions in the main program store an address that the subroutine can use to compute the address to which it should return when it is finished. In most computers there is a special instruction that facilitates this storage of the return address. In the 1401 there is an optional instruction called Store B Address Register, which, in conjunction with a special aspect of the 1401 index registers, makes it a simple matter to obtain the address of the next instruction after a Branch. The first instruction of the subroutine can store this Branch address at the end of the subroutine. No matter where the subroutine was entered from, therefore, the subroutine will return to the next instruction after that. In the absence of this special feature, it is not difficult to do essentially the same thing with standard instructions.

We see now the contrast between an open subroutine, which is inserted where needed and as many times as needed, and a closed subroutine, which is inserted in the program once and to which the main program branches whenever it is needed. Subroutines are used for two reasons: to save the trouble of writing routines that are already available (this applies both to open and closed subroutines) and to save storage space (this applies only to closed subroutines).

There are available for most computers a group of routines that come into this area of discussion although they are not set up as subroutines. Examples are programs to load cards, clear storage, print out specified areas of storage for help in program checkout, etc. For machines where the primary input is through punched cards, these are available on small decks that are readily available at the computer. At least a few of them will generally find use in virtually every program that is run on the machine. The name *utility routines* is applied to a broad category of programs of this type.

In the case of the 1401, there are three heavily used utility programs that illustrate this concept. The *clear storage* program is a two-card routine that clears all of storage to blanks, removes all word marks and then sets a word mark in location 001. It is typically placed at the front of every program loaded into storage to insure that each program begins with a clean slate. It is therefore unnecessary for each programmer to write clear storage instructions at the beginning of his program. The *card loader* is also a two-card routine. It will accept cards of the type produced by the SPS assembly program and load the instructions or constants punched on them into the specified locations in storage. The program also sets all word marks required by the instructions or constants. Finally, the card loader recognizes the card in the object deck produced by the END card in the SPS assembly, and branches to the location in the object program specified by this card.

A complete object program deck is typically organized as follows:

Clear Storage
Card Loader
Object Program Deck
Transition Card (produced from END card)
Data Cards

The loading of the entire program is accomplished by pushing the *load button* on the 1401 *console*. This button automatically causes a word mark to be set in 001, the first card to be read into the read storage area, and the instruction at 001 to be executed. The clear storage routine is set up on its two cards so that these actions will enable it to get started properly. From this point on, all card reading is initiated by instructions in the two utility programs and, later, in the object program. Thus, the clear storage routine loads the card loader program after having cleared all of storage. The card loader program loads the

object program deck. When the transition card is read, the loader program causes a branch to the object program, which then reads and processes data cards.

The last utility program that we will consider is one that prints a specified area of storage. This is typically used in checking out a new program when it is desired to see what the storage contents are after attempting to run it. The programmer punches on a *control card* the beginning and ending addresses of the region of storage that he wants printed. This control card is added at the end of the print storage deck and the deck loaded. The print storage program then prints out the contents of the specified storage locations, using an extra printing line to print 1's underneath the characters in which word marks are set. The whole operational sequence of punching a control card, loading the print storage deck and printing out the contents of all of storage can be done in a matter of a few minutes, giving extremely valuable data for use in determining whether the program is operating correctly and in establishing what is wrong with it if it is not.

Exercises

*1. The following fields are in storage:

Field	Symbol	Length	Sample
Date	DATE	5	05 (Month) 22 (Day) 1 (Year)
Account Number	ACCT	7	0078405
State	STATE	4	OREG
Amount Due	DUE	7	0164329

These fields are to be printed as shown in the following sample:

bb78405	OREG	b5	22	1	b1,643.29
1-7	11-14	18-19	21-22	24	28-36

Write a program segment to set up this printing line. The four fields have word marks in their high-order positions only.

2. The following fields are in storage:

Field	Symbol	Length	Sample
Name with two initials at right	NAME	22	JOHNSONbbbbbbbbbbbbRB
Social Security Number	SS	9	535221583
Amount	AMNT	6	86189

These fields are to be printed as shown in the following sample:

RbBbJOHNSONbbbbbbbbbbbb	535-22-1583	$ 861.89
1-24	28-38	42-49

Write a program segment to set up this printing line. The three fields have word marks in their high-order positions only.

*3. A deck of cards contains, among other things, an account number in columns 1-5 and a dollar amount in 23-28. The deck is in sequence on account number, and there are never more than 40 cards having the same account number. All the cards for one account number are to be printed on a separate page. The account number should be printed with zero suppression in positions 1-5, and the amount with a decimal point and zero suppression in 10-16. When all the cards for one account have been printed, a line should be skipped and the dollar total for the account printed with a decimal point and zero suppression in 8-16.

Draw a block diagram and write a program.

4. Two fields from each card in a deck are to be printed. Columns 1-7 contain an account number that is to be printed in positions 1-7 with zero suppression. Columns 8-14 contain a dollar amount that is to be printed in positions 11-20 with dollar sign, comma, decimal point, and zero suppression. A heading is to be printed at the top of each page, consisting of ACCOUNT in 1-7 and AMOUNT in 13-18. After 40 body lines, a line is to be skipped and the total of all of the amounts on the page printed in edited form in 9-20. When the last card is detected, print the total for the partial page and skip to the top of the next page. Draw a block diagram and write a program. Use either a line counter or a page overflow punch in channel 12 to detect the end of each page. (*Hint:* Be sure your program doesn't fall apart if the last page contains exactly 40 lines.)

5. Compare the total input and output time, and the time available for processing, for:

a. Reading a card, punching a card, and then printing a line (without read release, punch release, or print storage).

b. Executing the Write, Read, and Punch instruction (without read release, punch release, or print storage).

c. Executing the Write, Read, and Punch instruction (without read and punch release but with print storage).

6. Estimate the time required to execute the program of Figure 2, Section 3, exclusive of reading and printing.

7. Estimate the time required to execute the program of Figure 1, Section 4:

a. *Once*, exclusive of reading and printing.

b. For 100 cards, including reading and printing (without print storage).

c. For 100 cards, including reading and printing (with print storage).

*8. Estimate the time required to execute the program of Figure 3, Section 5, for a deck of 10,000 cards, including reading and printing (without print storage). Assume five cards per group. (*Hint:* You might begin by deciding whether it is worth worrying about the time for housekeeping, or about the alternative paths in the program—since they may or may not have any significant effect on total time. An estimate

within 5% is pretty good.)

9. Estimate the time required to execute the program of Figure 4, Section 6, for a deck of 2,000 cards (with print storage). (See hint on exercise 8.)