Principles of Programming

Section 6: Address Modification and Loops

IBM Personal Study Program

© 1961 by International Business Machines Corporation

Section 6: Address Modification and Loops

6.1 Computations on Addresses

We have seen in previous sections that instructions are stored within a computer in much the same way data is stored. An instruction is made up of the same characters that are available for storing data, the instruction characters are placed in the same storage as data, and in the 1401 instructions are required to have word marks in their high-order positions just as most data words have. As long as an instruction is simply being stored, it is literally indistinguishable from data. It is only when an instruction is to be *executed* that any differences arise. The fact that in the 1401 instructions are brought from storage in a left-to-right fashion, whereas data is accessed from a right-to-left, is really only a matter of design convenience and is not fundamental.

The one thing that actually distinguishes an instruction from data is the *time* at which it is brought from storage—that is, during the instruction phase or during the execution phase. If a word is read out of storage during the instruction phase, it goes to the control registers and is treated as an instruction. If a word is read out of storage during the execution phase, it goes wherever the operation code dictates that it should go to execute the processing prescribed by the instruction. This distinction between instructions and data is the same for all stored-program computers. It does not depend on the fact that the 1401 has a variable word length, or that word marks are involved, or that most instructions have two addresses, or any of the other features of the 1401 that are not typical of all computers.

What significance does all of this have to us as programmers? In a nutshell, the answer is that we are enabled to operate on instructions in storage just as though they were data. If one instruction says to add a constant to the address of another instruction, there is no confusion in the machine in doing so. The first instruction, which calls for the addition, is accessed during the instruction phase and goes to the control registers. The address part of the second instruction, on which arithmetic is being performed, is accessed during the execution phase of the first instruction. Similarly, if a Move instruction is used to transfer an instruction from one place in storage to somewhere else, this is perfectly legitimate. It is also permissible to have one instruction change the operation code of another instruction.

The facility for carrying out processing operations on instructions is one of the most important aspects of a stored-program computer. In short, it makes it possible to set up a program to *modify itself*, according to the results of its own data processing operations. This ability, combined with the ability to repeat a section of a program that is provided by the various branching instructions, is by all odds the most important single feature of the stored-program concept.

For a first example of the application of this concept, consider the following sales summarization problem. A previous computer run has produced a deck of cards containing (among other things) a sales amount and a code number that gives the class of merchandise represented by the sale. There are six classes of merchandise, represented by the codes 1 through 6. The merchandise code is punched in column 43 and the amount of sale in dollars and cents is punched in columns 17 to 22. In this highly simplified example, we are required only to print in a single line the total sales of each merchandise class. The required printing positions are:

Merchandise Class	Printing Position
1	1-10
2	11-20
3	21-30
4	31-40
5	41-50
6	51-60

This example presents only one problem: How to determine to which of the six total accumulators the sales amount on each card should be added. The reading of the cards, the last-card test and the printing of the total line will cause us no difficulty. The choice of the proper accumulator *could* be handled by a series of comparisons and branches, or somewhat more conveniently by the use of an instruction that we have not considered, the Branch If Character Equal instruction. However, by either of these methods, the testing and branching would run to 15 or 20 instructions—which makes us wonder whether there might not be some simpler way to accomplish the same result.

Indeed there is a simpler way. Consider what we would get if we were to multiply the merchandise code by 10 and add the product to 200:

Merchandise Code	$10 \times$ Code + 200
1	210
2	220
3	230
4	240
5	250
6	260

Thus it appears that if we carry out this simple computation on the merchandise code and use the result as the address of an instruction, then in each case we will have the address of the proper field in the print area. Since the entire computation produces only one line of output, there is no reason not to use the print area itself for the six accumulators. Therefore, once the address of the proper position in the print storage area has been computed, it can be placed in the address part of an Add instruction and the addition performed in *whichever accumulator is specified by the computed address.*

Since we are concerned in this example with other things, the program in Figure 1 is shown without the initial housekeeping operations of clearing storage and setting word marks. After reading a card, we proceed immediately to compute the address of the accumulator to which the sales amount from this card should be added. The address of this accumulator will be developed in a three-position field which has been given the symbolic address WKSTOR, for working storage. We begin by moving the constant 200 into this working storage. Then the merchandise code is added to this field with character adjustment of minus one, which has the effect of adding the code into the tens position of the field. Therefore, the sum in WKSTOR will be 200 plus ten times the merchandise code, which as we saw is the address of the proper accumulator. This address is next moved into the B-operand address part of the Add instruction, which follows immediately. This is done with the MCW instruction in which the B-operand address is COMPAD with character adjustment of plus six. Looking at the labels in this program, we see that COMPAD is the symbolic location of the Add instruction. Remembering that the location of an instruction refers to the location of the operation code, we see that to obtain the address of the rightmost character of the B-operand does require character adjustment of plus six.

The Add instruction which will form the sum is shown with a B-address of 0000. This is to remind us that this address is computed by the program itself. The situation is this: When the object program is loaded into storage, the B-address of this instruction is 0000. However, *by the time the instruction is executed*, the program itself will have placed the address of one of the six accumulators in this part of the instruction.

With the sales amount added to the proper accumulator, we make a last-card test. If this was the last card we branch to print the total and halt; if it was not the last card, we branch back to read another card and repeat the entire process.

It is important to realize just what this example shows: that computations on addresses are possible and useful. This particular example, however, is unrealistic—for a reason that is important in itself. What would happen if a merchandise code were mispunched, and entered the

IBM

1401 Symbolic Programming System
Coding Sheet

Program ____________ Page No. 1 of 1
Programmed by ____________ Date ________ Identification ________

LINE	COUNT	LABEL	OPERATION	(A) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	(B) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	d	COMMENTS
010		READ	R										
020			MCW	C200				WKSTØR					ADDRESS -
030			A	CØDE				WKSTØR	-	001			CØMPUTATIØN
040			MCW	WKSTØR				CØMPAD	+	006			STØRE ADDRESS
050		CØMPAD	A	SALE				0000					CØMPUTED B-ADDRS
060			B	PRINT								A	LAST CARD TEST
070			B	READ									NØ
080		PRINT	W										PRINT TØTALS
090			H	*	-	003							
100			H										
110	03	C200	DCW	*		200							
120	03	WKSTØR	DCW	*									
130		CØDE	DS	0043									
140		SALE	DS	0022									
150			END	READ									
160													
170													
180													
190													
200													

Figure 1. Program illustrating address computation.

computer as 7 or K? The answer is simply that the program as written would carry out the address computation on the bad code, and then add the sales amount to whatever location it computed. The result would be at least wrong—and perhaps disastrous: the program might well be destroyed by the addition.

The point of all this is that one should think twice before putting so much faith in data. In this example, we could make a check before the address computation to determine that the code really is a digit between one and six. Different ways to guarantee the correctness of a computed address may be found in other situations.

In any case, the example illustrates well the principle of address computation, and perhaps gives a hint of the usefulness of the technique.

Review Questions

1. *How does a computer distinguish between instructions and data?*
2. *Would the concept of storing instructions like data and the consequent ability to perform arithmetic on instructions be significantly different in a machine in which instructions have three addresses?*
3. *Why was it necessary to develop the address of the correct accumulator in a working storage area rather than directly in the address part of the Add instruction? (Hint: Consider what would happen when the second and subsequent cards were read, and what the word mark problems might be.)*
4. *This particular program is completely dependent on the fact that each of the printing fields is exactly ten columns long. Still using the same basic address computation technique, how could you change the program if each of the printing fields were 15 columns long?*

6.2 Program Switches

For another example of the concept of a program modifying itself, consider the "storage" of decisions, through the use of program switches. It not infrequently happens that a decision made at one point in the program has a bearing at one or more later points. Sometimes it is possible simply to repeat the branch instruction that made the decision in the first place. In other cases, however, the information on which the decision was originally made is no longer available—or it may happen that the decision involves a number of instructions, making it wasteful to repeat them when the result of the decision is needed later.

When such situations arise, it is desirable to be able to store the result of the decision. This can be done in many ways. One possibility is to store either a zero or a one in some location, depending on the outcome of the test. Then when it is later necessary to know what the result of the test was, this storage location can be checked to see whether it contains a zero or a one. The most common technique, however, at least for storing the results of two-way decisions, is to change the operation code of instructions.

This instruction modification is most frequently done using the unconditional Branch and the No Operation instructions. No Operation is an instruction that causes no action to take place anywhere in the computer. Stated otherwise, there is no execution phase on this instruction. It is provided partly for such situations as we are describing, and partly to make it possible to eliminate the effect of unwanted instructions when it is not feasible to reassemble. It has many other valuable uses. The operation code is N and the instruction may have any of the other parts of the instruction; any other parts besides the operation code will, of course, have no effect.

To describe the operation of program switches a little more concretely, consider the following situation. A comparison is to be made early in a program. If the comparison shows equal, then at three subsequent points in the program it is necessary to branch to special routines to handle this case. If the comparison shows unequal, then at each of those three points the program should continue in sequence. The technique is to write the three Branch instructions at the points where the program should transfer out to the special routines, as though the branch would always occur. When the test is made, one of two short routines is executed. If the test shows unequal, then the operation codes of the three Branch instructions are changed to N. If the test shows equal, the three operations codes are set to B. When the three instructions are subsequently executed, they will either cause the Branches or allow the program to continue in sequence, depending on what the result of the test was.

It actually is necessary to go to the trouble of setting the operation code of the three switches for *each* outcome of the test. It might be thought that if the instructions are originally written as Branches, then they could simply be left alone if the initial test shows that the branch should be executed, and changed to N's if the program should continue in sequence. This would indeed work correctly the first time through the program and possibly for a few later executions. However, as soon as the operation codes are once changed to N, then they need to be reset to B's if the Branch should be executed.

Naturally there are many other programming techniques that can be used to store the result of a decision. One that comes to mind immediately is the possibility of changing the address part of a Branch

instruction. The choice of the method to be used in setting up a program switch depends on such factors as how many different possible outcomes the decision has, how many places the switch must operate, how much trouble it is to repeat all or part of the original decision, etc. In other computers, the choice will also depend on the programming characteristics of the machine.

In all cases, however, the general principle is simply that a decision made at one point in the program is being used to control the subsequent action of the same program on one or more later occasions. This further example of the modification of a program by itself finds fairly frequent application in many programs. We shall see a few examples of the technique in later sections.

Review Questions

1. *Describe how a program switch could be set up using modification of the address of a Branch instruction.*
2. *Does the concept of a program switch depend on using the result of a decision at* more than one *subsequent point in the program?*

6.3 Program Loops

It must be readily apparent that a program involving no repeated executions of instructions would not be practical. If a program were only able to proceed sequentially through its instructions and upon completion had to be replaced by another program, then it is clear that the stored program computers would be of little value: most of the time would be spent in loading instructions.

Fortunately, however, there is no such restriction on the organization of programs, and a whole body of technique has been built up around the methods for repeated execution of program segments. This technique is known as *looping*. We have, in fact, already seen a number of elementary examples of loops. The illustrative program in Section 5.1 is a loop in the following sense: After some preliminary housekeeping operations, we read a card and perform certain computations on the data read from it. Then we test the last card indicator to determine whether all of the cards have been read. If not, we return to the instruction for reading a card and repeat the entire program except for the initial housekeeping operations.

We have here almost all of the normal parts of a loop. There is an *initializing* section, which gets the loop started properly and is only executed once. There is a *computation* section which does the actual work of the program—in this case, reading a card and performing the

calculations. There is a *testing* section which determines whether the work of the loop is completed. Most loops also contain a *modification* section which changes some of the instructions in the computation section of the loop or changes the data on which the computation section operates. In a certain sense, even the simple program in Section 5.1 has a modification section if we regard the reading of a new data card as a modification of the data being operated upon.

This example is a loop that consists of an entire program, which is a rather broad application of concept. We more commonly find loops which are only small segments of a total program. Frequently, one loop has within it one or more additional loops. The sales summarization program of Section 1.3 can be viewed in this manner. The innermost loop is the one that obtains the sales total for each salesman. This loop is "inside" the loop that computes the totals for each district and this, in turn, is "inside" the total program loop that reads the entire deck of sales cards. In each case there is an initialization section. This consists of the housekeeping operations for the total program loop, and of the special handling of the sales amount on the first card of each group for the other two loops. In each case, there is a computation section; this must be applied broadly in the case of the total program loop since it consists of all of the operations contained in the other two loops. In the two summarization loops, the computation consists of the summarization and of the processing that is done when it is found that the last card of a group has been read. In each case there is testing, in one case to detect the last card (although this test was not written in the program earlier), and in the other two cases to determine when the first card of a new group has been read.

The loop concept provides the best example of the unique power of a stored program digital computer. It is probably the most important single topic in the study of programming. We shall see immediately below that one of the most powerful types of loops involves the repetitive modification of the instructions within the loop itself, most commonly the addresses.

Review Questions

1. *Name the four parts of a loop and give examples of each.*
2. *Must the four parts of a loop always be executed in the order in which they are named in the text?*
3. *If a loop is only used once in a program, that is, never started again with new data, is it logically necessary to initialize?*

6.4 Address Modification Loops

In this type of loop, we have as before the four parts of initialization, computation, testing and modification, although not necessarily always in that order. The modification now consists of changing one or more addresses within the computation section of the loop. The testing most commonly involves determining whether the computation section has yet been carried out a specified number of times.

For an example of this type of loop, consider the following inventory usage application. A deck of cards contains one card for each part in the inventory of a certain manufacturing company. Each card shows the part number and the usage for each of the twelve months of the calendar year. The task is to produce a report with one line for each part, showing the average monthly usage of the part and the number of the month in which maximum usage occurred.

The card format is as follows:

Columns	Field
1-8	Part No.
9-13	January Usage
14-18	February
19-23	March
24-28	April
29-33	May
34-38	June
39-43	July
44-48	August
49-53	September
54-58	October
59-63	November
64-68	December

The format of the report is as follows:

Printing Positions	Field
1-8	Part Number
12-16	Average Monthly Usage
20-21	Number of month of maximum usage

This program may be thought of as consisting of two parts: dividing the sum of the monthly usages by 12 to get the average, and determining which of the months has the heaviest usage. In the final version of the program these two parts will be combined in one loop. However, to get a clear picture of the workings of an address modification loop we shall first write a program to get the average only, and later add the instructions for finding the heaviest usage.

After reading a card, the object of the summing loop is to add to an accumulator the usage for each of the twelve months. This of course

could be done with a Move and eleven Adds. However, we shall see that it can be done with fewer than twelve instructions, in the following manner: We begin by setting the accumulator to zeros in order to remove the sum developed there from the last card. We also set to zero a two-position counter that will be used to determine when the last monthly usage has been added to the accumulator. The twelve monthly usages are picked up from the read area by a single Add instruction, the address of which is modified each time through the loop. Since after reading one card this address will be incorrect for starting the accumulation of the usages from the next card, we initialize this address by setting it to 13, the address of the first data field.

Each time around the loop another data field is added to the accumulator, the A-address of the Add instruction is increased by 5, and 1 is added to the counter. A comparison is made each time to see whether this counter has yet reached 12. If it has, then all twelve monthly usages have been added into the total and we are finished; if it has not, then the loop is repeated. When the total usage for the year has been developed, we divide by 12, print the line for this inventory item, make a last-card test and, if cards remain, return to the Read instruction.

A block diagram of this procedure appears in Figure 2 and a symbolic program in Figure 3. Once again, the program is shown without the preliminary housekeeping operations of clearing storage and setting word marks. In the program a new instruction is used to place zeros in the accumulator and count fields. The Zero and Add instruction is just like an Add except that the B-field is cleared to zeros before the addition takes place. This instruction is therefore analogous to a Move or Load instruction, with the significant difference that in a Zero Add the zone bits of all but the low-order character are removed during the transmission. This feature itself is often of value. In our case, the advantage of a Zero Add over a Move is that we can set up a field consisting of only a single zero and clear the *entire* B-field. What the instruction actually does, in our case, is to clear the entire B-field to zero and then add the one-character constant of zero that we specify with the A-address.

Zero and Add

	Mnemonic	Op Code	A-address	B-address
FORMAT	ZA	?	xxx	xxx

FUNCTION The entire B-field is set to zeros, then the data from the A-field is moved to the B-field with zone bits stripped from all but the units position. If A is shorter than B, zeros are placed in the high-order positions of B.

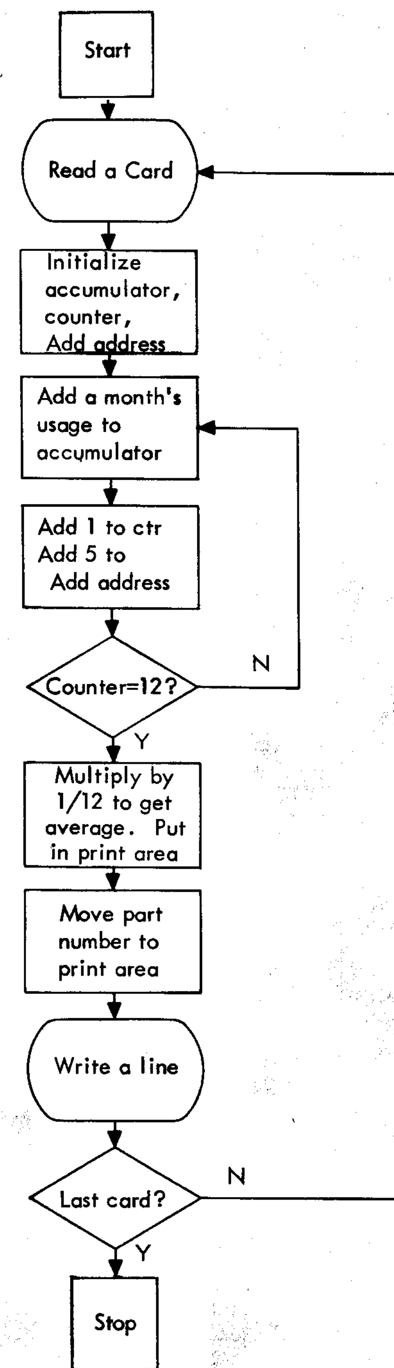

Figure 2. Block diagram of a procedure to compute the average of twelve numbers on a card.

Page No. 1 of 2

LINE 3–5	COUNT 6–7	LABEL 8–13	OPERATION 14–16	(A) OPERAND ADDRESS 17	± 23	CHAR. ADJ.	IND. 27	(B) OPERAND ADDRESS 28	± 34	CHAR. ADJ.	IND. 38	d 39	COMMENTS 40–55
010		START	R										
020			ƵA	ƵERØ				ACCUM					INITIALIƵE
030			ƵA	ƵERØ				CØUNT					TØTAL & CØUNTER
040			MCW	INTADD				ADDINS	+	003			INIT ADDRESS
050		ADDINS	A	0000				ACCUM	−	007			VARIABLE ADDRESS
060			A	ØNE				CØUNT					MØDIFY CØUNT
070			A	FIVE				ADDINS	+	003			& VAR ADDR.
080			C	CØUNT				TWLVE					FINISHED Q
090			B	ADDINS								/	NØ
100			M	CI				ACCUM					YES MULT 1/12
110			A	FIVE				ACCUM	-	005			RØUND
120			MCS	ACCUM	-	006		0216					SET
130			MCS	0008				0208					UP
140			W										AND PRINT
150			B	LAST								A	LAST CARD Q
160			B	START									NØ
170		LAST	H	*	−	003							YES
180			H										
190													
200													

Figure 3. Program to compute an average, as diagrammed in Figure 2

Page No. 2 of 2

LINE 3–5	COUNT 6–7	LABEL 8–13	OPERATION 14–16	(A) OPERAND ADDRESS 17	± 23	CHAR. ADJ.	IND. 27	(B) OPERAND ADDRESS 28	± 34	CHAR. ADJ.	IND. 38	d 39	COMMENTS 40–55
010	14	ACCUM	DCW	*									14 CHAR FOR MULT
020	02	CØUNT	DCW	*									
030	01	ƵERØ	DCW	*		0							
040	03	INTADD	DCW	*		013							
050	01	ØNE	DCW	*		1							
060	01	FIVE	DCW	*		5							
070	06	CI	DCW	*		083	3	33					
080	02	TWLVE	DCW	*	+	12							
090			END	START									
100													
110													
120													
130													
140													
150													
160													
170													
180													
190													
200													

Figure 3. Continued

WORD MARKS The B-field must have a word mark; the A-field must have a word mark only if it is shorter than the B-field.

TIMING $T = .0115\ (L_I + 1 + L_A + L_B)$ ms.

The desired initial address of the Add instruction that picks up the monthly usages is transferred with an MCW instruction.

The variable-address Add instruction, which has the symbolic label of ADDINS, is shown with an A-address of 0000; this address is computed by the program and will have some value other than 0000 by the time it is first executed. After the first monthly usage is added into the accumulator with character adjustment to add into the high-order part, we add a 1 to the counter and add 5 to the variable address of the Add instruction. This last is done with character adjustment to add the 5 into the units position of the A-address. It might appear that there is a word mark problem here, but it happens that the attempt to propagate carries when the 5 is added to the address will not affect the operation code of the Add instruction. If it were desired to be double safe on this, a word mark could be set in the high-order position of the A-address before the addition of the 5 and then cleared afterwards.

Next the count is compared with 12 and a Branch If Indicator On instruction tests for equality. If the indicator shows that the two are unequal, we branch back to the Add instruction and pick up another monthly usage and continue the loop. If the unequal indicator is off, then the branch does not occur and we proceed to find the average. This could be done with the Divide instruction, an optional feature on the 1401, or by a programmed division routine. Here, however, we have chosen to multiply by 1/12 rather than divide by 12. This is done simply to save the time that would be required to describe division in the 1401, since we shall have no further occasion to use it.

The constant 1/12, which is taken as equal to .083333, has six places to the right of the decimal point. Therefore, after the multiplication the average can be rounded to the nearest unit by adding a 5 to the fifth digit to the left of the units position. The rounded monthly usage is then moved with zero suppression to the printing position, the part number is also moved with zero suppression, the line printed, and a last-card test made.

It is worth emphasizing what this program illustrates. We have here a fairly representative example of an address modification loop. There is an initializing section, where we put zeros in locations that could have left-over data from the previous execution of the complete loop, and where we start an address at its correct initial value. There is a computation section, consisting in this loop of just the one variable-address Add instruction. The modification section consists of the addition of 1 to the counter and of 5 to the address of the Add instruction. The testing involves determining whether the counter has reached 12 and returning to another execution of the loop if not. The instructions that follow the test are not part of this loop.

Notice that the complete loop takes eight instructions, including the initialization. Without a loop, the same summation would take twelve instructions: one MCW and eleven Adds. However, the loop version requires the execution of 63 instructions: three for initialization, and twelve times around the five instructions in the repeated portion of the loop. Thus we see that a loop saves space at the expense of time. This is a completely general statement.

This example is quite important for what it shows about the way computers are programmed. The student is urged to understand this example thoroughly before proceeding.

With this basic loop clearly understood, we can without too much difficulty extend it to include finding the month having the largest usage. This can be done by starting with the initial assumption that January has the largest usage. Then 1 is stored in a location that will contain the number of the month having the largest usage. January's usage is then compared with February's; if January's is larger, then January is still the largest of those considered so far and the 1 is maintained as the number of the month having the largest usage. If, on the other hand, February has larger usage, then February's usage is moved to the location containing the largest usage so far and the 2 is placed in the location containing the number of the largest month. This "largest usage to date" is continually compared with each succeeding month and either left where it is if it is larger or replaced by the other month if that one is larger.

In order to simplify the loop, what we will actually do is begin by comparing January's usage with itself. Thus we avoid having to set up a somewhat longer initializing section to get the loop properly started, bearing in mind that along with this testing we are still accumulating the total usage in order to develop the average. This may seem like a waste of time, which it is, but it is worth it: we are saved the complication and the space that would be required to make the loop operate differently the first time. This also is a rather general situation. Simplicity is usually a virtue in programming, since it reduces the likelihood of making mistakes. Furthermore, the time that might be saved by repeating the loop one less time would probably be completely offset by the extra instructions that would be required to get it started properly.

A block diagram is shown in Figure 4 and a symbolic program in Figure 5. In this complete program there are a number of additional things to initialize. Besides the accumulator and the counter, there are now three variable addresses to start properly. January's usage must be moved to LARGE and a 1 must be put in the month number. This last is transferred with a Zero Add to get the one-digit constant of 1 into the two-digit field.

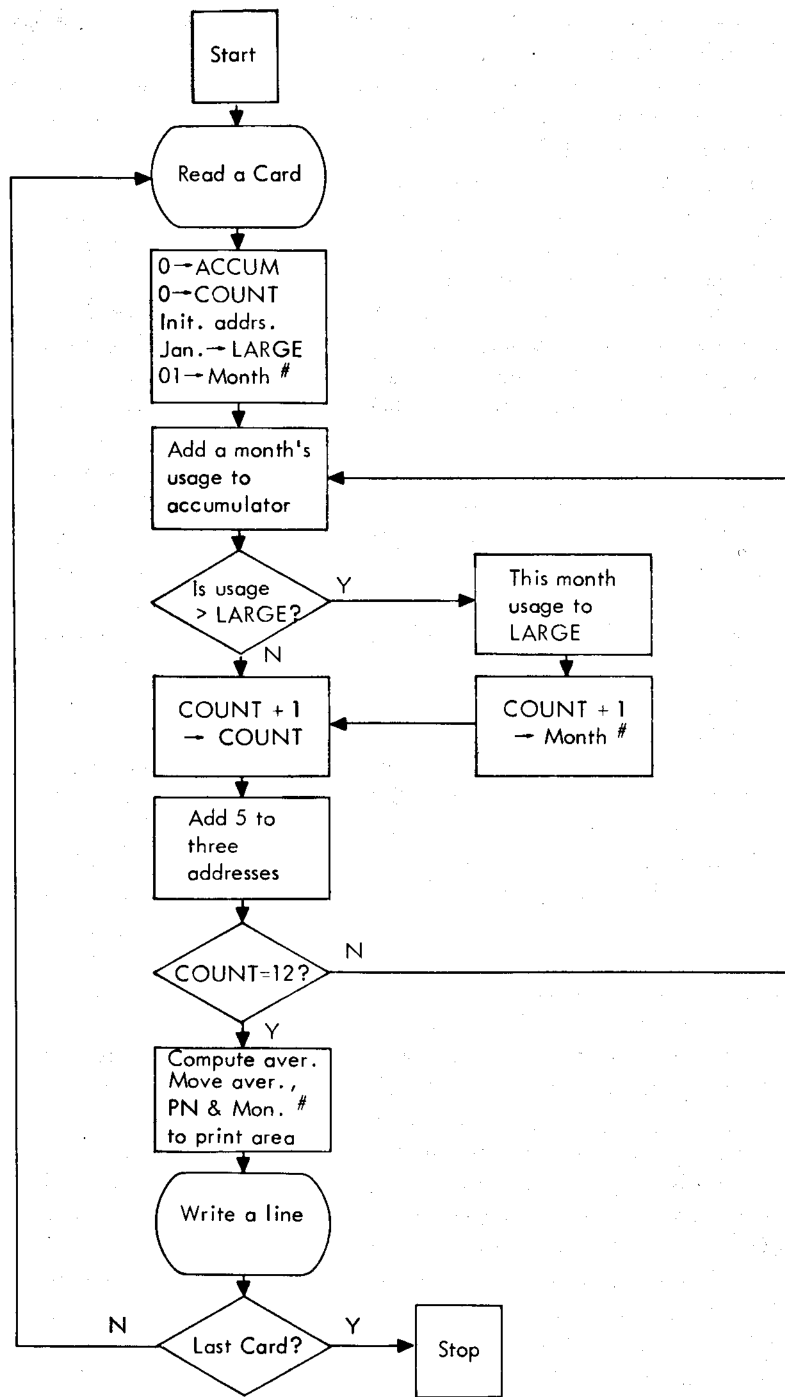

Figure 4. Block diagram of a procedure to compute the average of twelve numbers on a card, and find which of the twelve is largest.

Page No. 1 of 2

LINE (3–5)	COUNT (6–7)	LABEL (8–13)	OPERATION (14–16)	(A) OPERAND ADDRESS (17–23)	±	CHAR. ADJ.	IND. (27)	(B) OPERAND ADDRESS (28–34)	±	CHAR. ADJ.	IND. (38)	d (39)	COMMENTS (40–55)
010		START	R										
020			ZA	ZERØ				ACCUM					INIT. TØTAL
030			ZA	ZERØ				CØUNT					& CØUNTER
040			MCW	INTADD				ADDINS	+	003			INIT.
050			MCW	INTADD				CØMINS	+	003			ADDRESSES
060			MCW	INTADD				MØVINS	+	003			X
070			MCW	0013				LARGE					JAN. USAG
080			ZA	ØNE				MØNTHN					MØNTH NØ. 1
090		ADDINS	A	0000				ACCUM	–	007			SUM - VAR. ADDRS
100		CØMINS	C	0000				LARGE					CØMP USAGE
110			B	MØDIFY								U	BR. IF LARGER
120		MØVINS	MCW	0000				LARGE					THIS MØ. LARGER
130			MCW	CØUNT				MØNTHN					MØVE NØ. TØ NØ.
140			A	ØNE				MØNTHN					ØF LARGEST MØNTH
150		MØDIFY	A	ØNE				CØUNT					INC. CØUNT
160			A	FIVE				ADDINS	+	003			MØDIFY
170			A	FIVE				CØMINS	+	003			ADDRESSES
180			A	FIVE				MØVINS	+	003			X
190			C	CØUNT				TWLVE					FINISHED Q
200			B	ADDINS								/	NØ

Figure 5. Program to carry out the procedure diagrammed in Figure 4.

LINE	COUNT	LABEL	OPERATION	(A) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	(B) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	d	COMMENTS
010			M	CI				ACCUM					YES. MULT. 1/12
020			A	FIVE				ACCUM	-	005			RØUND
030			MCS	ACCUM	-	006		0216					SET
040			MCS	0008				0208					UP
050			MCS	MØNTHN				0221					AND
060			W										PRINT
070			B	LAST								A	LAST CARD Q
080			B	START									NØ
090		LAST	H	*	-	003							
100			H										
110	14	ACCUM	DCW	*									
120	02	CØUNT	DCW	*									
130	01	ZERØ	DCW	*		0							
140	03	INTADD	DCW	*		013							
150	01	ØNE	DCW	*		1							
160	01	FIVE	DCW	*		5							
170	06	CI	DCW	*		083	3	33					
180	02	TWLVE	DCW	*	+	12							
190	05	LARGE	DCW	*									
200	02	MØNTHN	DCW	*									
			END	START									

Figure 5. Continued

The variable address Add instruction is as before. We next compare the current month's usage with what is so far the largest usage. The first time through this will compare January with January, but no damage is done and a few instructions are saved. If the one that has been largest so far is larger than the current month's usage, we branch directly to the modification section. If it is not, the current month's usage is moved to LARGE, the count is moved to the month number and one is added to this count. This is necessary because the count as set up here is always one less than the number of the month with which we are currently dealing.

The modification section is just about as before except that there are three addresses to modify. The final instructions of the program are the same except that the new field must also be moved to the print area.

Review Questions

1. *Suppose the COUNT had been initialized to 1 instead of zero. What constant would have to be changed?*
2. *Suppose the COUNT had been tested* before *adding 1 to it. What should the COUNT be tested against in this case?*
3. *What would happen if a Zero and Add instruction were used to transfer alphabetic data?*

6.5 Indexing

We see in the program just completed that a fair number of instructions were used in doing nothing but modifying addresses. In many programs, a fairly high fraction of the total instructions are involved in operations that are required to get the program to operate correctly but which do not themselves directly process any data. A valuable machine feature in reducing this kind of red tape is the indexing of addresses.

The basic idea of indexing is to leave the addresses of the variable address instructions unchanged as they appear in storage, and to modify them with the contents of an *index register* each time they are executed by the object program. Between executions of the repeated portions of the loop, we can change the contents of the index register. This will have the effect of changing the effective address but will not actually change the instruction as it appears in storage. This is because the addition of the index register contents to the address as written is carried out in the address registers and not in storage. To summarize: Instead of actually changing the addresses of instructions that vary, we instead specify that before execution the address as written should be

incremented by the contents of an index register. This process does not change the instruction as it appears in storage; we can get the effect of a variable address simply by changing the index register contents.

In a loop in which only one address has to be modified, this procedure does not offer any strong advantages unless there are specialized instructions for doing combination operations on the index registers. Even in the absence of such features, however, the indexing principle becomes very valuable if there are several instructions that have to be changed, since the same index register can be used to modify any number of instructions. The initialization now consists of just the one instruction required to put the proper initial contents into the index register, and the modification consists only of adding the required constant to the index register. Furthermore, the index register now also serves as a counter that can be used to determine when the loop operation is completed.

In the 1401 there are three index registers, which are named 1, 2 and 3. Index one consists of storage locations 087-089; index two 092-094; index three 097-099.

To add the contents of an index location to the address of an instruction, we tag the address that should be modified. This is done, in actual machine language, using the zone bits of the tens position of the address, in the following pattern:

Index Location	Tens Position Zone Bits	Zone Punch
1	01	Zero
2	10	Eleven
3	11	Twelve

On the symbolic programming sheet, it is necessary only to write the number of the desired index location in the appropriate IND column—that is, column 27 or 38.

When an indexed instruction is executed, the sequence of operations within the machine is as follows. The instruction is first brought to the control section registers just as it always is. During this process, the zone bits of the tens position will be detected as specifying indexing. The contents of the specified index location are then obtained from storage and added to the contents of the address register. The instruction is then executed. Notice that the instruction as it appears in storage is not changed by indexing. (The index locations are, of course, not changed either.) The address as modified by the contents of an index location is called the *effective address.*

In order to change the effective address, it is necessary only to change the contents of the index location, which may be done with ordinary 1401 instructions. To do this, the index locations will ordinarily have to have word marks, since the locations are not treated any differently from any other locations in storage, except as they are called on by the execution of an indexed instruction. When the index locations are not being used for indexing they may be used for any other purpose.

We may see how indexing can be used by rewriting the program of the last subsection. The basic logic will not be appreciably different. There will be fewer instructions, because by initializing the one index location we initialize the *effective* address of the three instructions that must have variable addresses, and one instruction that adds 5 to the index changes the effective address of all three. Furthermore, the index can also be used as the counter. We will write the instructions that are to have variable *effective* addresses, with *actual* addresses of 0013. The index location, which is chosen to be 1 in the program shown below, is initialized to zero. Each time through the loop, 5 is added to this location; loop testing consists of asking whether index 1 contains 60.

The block diagram for this program is shown in Figure 6 and the program in Figure 7.

In this particular program it is still necessary to have a counter that counts by ones to know the month number as we make the comparisons to find the month having the largest usage. In this particular case it would not matter much whether the loop testing were done using the index register or by comparing this month counter against 13. In many problems, of course, there would not be this choice. We see that even though it is necessary to have what amounts to two loop counters, the program is still somewhat shorter than the unindexed version. We note that the three instructions that have variable effective addresses were written with actual addresses of 0013 and that index 1 is specified in column 27 in each case. It happens not to be necessary here, but it is also permissible to index B-addresses.

This example nicely illustrates the power of the indexing technique in reducing the auxiliary operations of an address modification loop. Since we are concerned primarily with the concept and not with the details of operation, we are omitting a complete description of what the machine does in certain situations involving addresses over 999, and a number of other matters that are important when using the 1401 but are not crucial to the indexing concept.

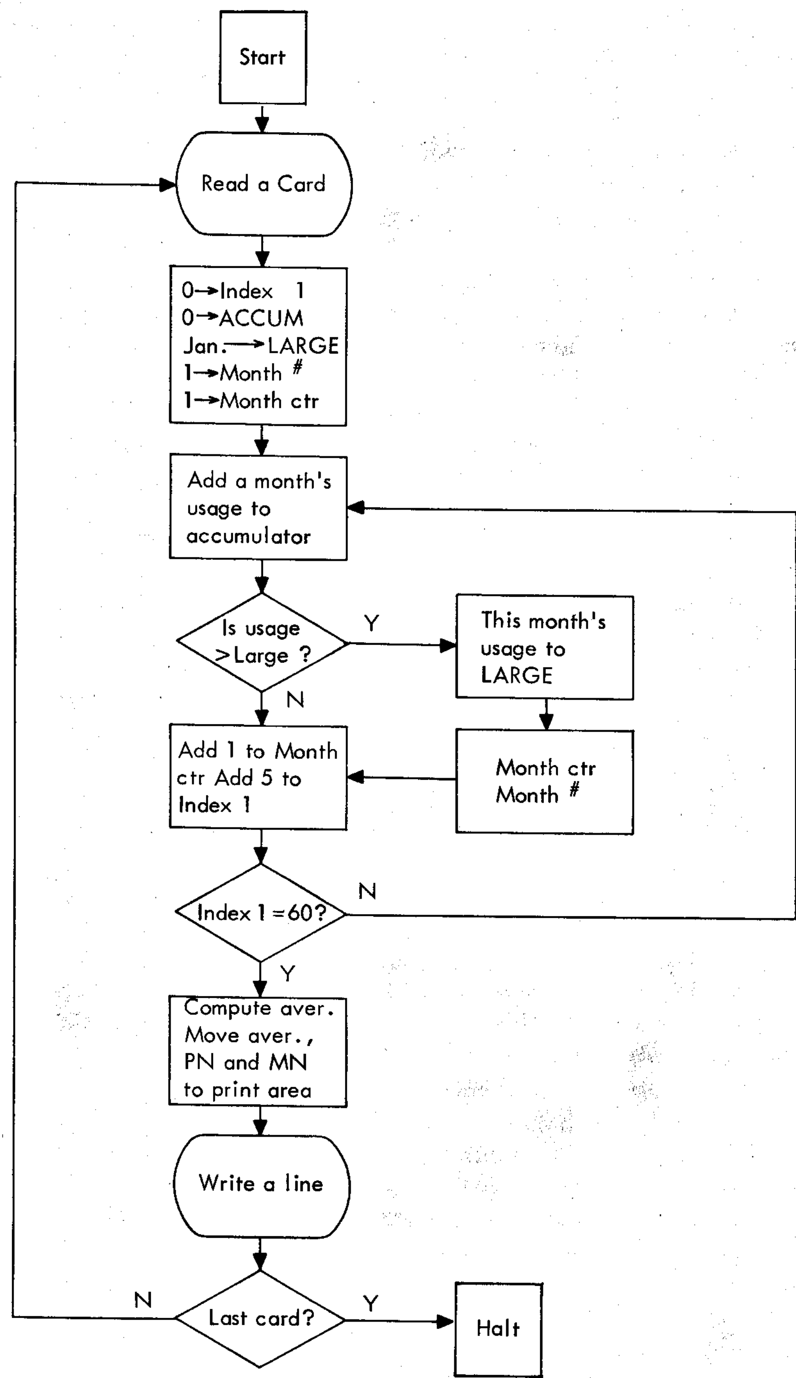

Figure 6. Figure 4, modified to show the use of indexing.

Page No. 1 of 2

LINE (3–5)	COUNT (6–7)	LABEL (8–13)	OPERATION (14–16)	(A) OPERAND ADDRESS (17)	(A) ± (23)	(A) CHAR. ADJ.	(A) IND. (27)	(B) OPERAND ADDRESS (28)	(B) ± (34)	(B) CHAR. ADJ.	(B) IND. (38)	d (39)	COMMENTS (40–55)
010		START	R										
020			ZA	ZERØ				0089					ZERØ TØ INDEX 1
030			ZA	ZERØ				ACCUM					ZERØ TØTAL
040			MCW	0013				LARGE					JAN. USAGE
050			ZA	ØNE				MØNTHN					MØNTH NØ. 1
060			ZA	ØNE				MTHCTR					1 TØ MØNTH CØUNT
070		ADDINS	A	0013			1	ACCUM	−	007			INDEXED
080		CØMINS	C	0013			1	LARGE					X
090			B	MØDIFY								U	BR. IF LARGER
100		MØVINS	MCW	0013			1	LARGE					INDEXED
110			MCW	MTHCTR				MØNTHN					NØ. LARGEST MØ.
120		MØDIFY	A	ØNE				MTHCTR					ADD 1
130			A	FIVE				0089					INDEX 1 + 5
140			C	0089				SIXTY					FINISHED Q
150			B	ADDINS									NØ
160			M	C1				ACCUM					YES. MULT. 1/12
170			A	FIVE				ACCUM	−	005			RØUND
180			MCS	ACCUM	−	006		0216					SET
190			MCS	0008				0208					UP
200			MCS	MØNTHN				0221					AND

Figure 7. The program of Figure 5, modified to use indexing.

Page No. 2 of 2

LINE	COUNT	LABEL	OPERATION	(A) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	(B) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	d	COMMENTS
010			W										PRINT
020			B	LAST								A	LAST CARD Q
030			B	START									NØ
040		LAST	H	*	−	003							YES
050			H										
060	14	ACCUM	DCW	*									
070	01	ZERØ	DCW	*		0							
080	01	ØNE	DCW	*		1							
090	01	FIVE	DCW	*		5							
100	06	C1	DCW	*		083	3	33					
110	05	LARGE	DCW	*									
120	02	MØNTHN	DCW	*									
130	02	MTHCTR	DCW	*									
140	02	SIXTY	DCW	*	+	60							
150	03		DCW	0089									
160			END	START									
170													
180													
190													
200													

Figure 7. Continued

Index registers, which are also sometimes called *B-boxes*, or *indexing accumulators*, are available on most computers. In some machines there are more than three, ten being a typical number. In some machines, the contents of the index register are subtracted from the actual address instead of added to it. In a number of computers there are specialized instructions that make indexing even more powerful. At least one computer has an instruction that is a combination of a conditional branch and a subtract, making it possible to write useful loops that have only two repeated instructions.

Review Questions

1. *Does indexing change the indexed instruction as it appears in storage?*
2. *Is indexing done in the processor or in the object program?*
3. *It is possible to have both character adjustment and indexing of a single address. Explain the effect of each, and when each is done.*
4. *What are the advantages of indexing?*

Exercises

•1. In the program of Figure 3, the variable address of the Add instruction could be used as a counter to determine when the loop has been executed twelve times. Rewrite the program accordingly. (*Hint:* Determine carefully what the loop testing constant should be.)

2. Modify the block diagram of Figure 4 and the program of Figure 5 to produce on the report the number of the month having the *smallest* usage as well as the largest. Write with or without indexing.

3. Modify the block diagram of Figure 4 and the program of Figure 5 to print an X behind the month number if two or more months had a usage larger than all others. Write with or without indexing.

•4. Draw a block diagram and write a program to do the following. Read a card and move columns 1-20 to 0401-0420; read another card and move columns 1-20 to 0421-0440; read another card and move columns 1-20 to 0441-0460, etc. When columns 1-20 of 20 cards have been moved to the new locations (the information from the last card goes to 0781-0800), leave a blank line in your program for writing the 400 characters in 0401-0800 onto magnetic tape. Then assemble another block of 400 characters in 0401-0800 and indicate writing on tape. Continue writing on tape until all cards have been read.

The number of cards in the deck is not necessarily a multiple of 20, so a last-card test must be made after moving each group of 20 characters. When the last card is detected, indicate the writing on tape of the last block, even though it is most likely not full.

Write with or without indexing.

5. Columns 21-22 of an invoice card contain a two-digit state number between 01 and 50. Write a program segment to find the four-character alphabetic abbreviation corresponding to the number and place the abbreviation in positions 0237-0240 in the print area. There is a table in storage, as follows:

State number	Address of abbreviation
01	0785
02	0789
03	0793
04	0797
.	.
.	.
.	.
50	0981

A loop is not necessary.