Principles of Programming

Section 5: Branching

IBM Personal Study Program

Section 5: Branching

5.1 Fundamentals of Branching

In the last section we saw an example of a branching operation, in connection with the last instructions of the two sample programs. These instructions were set up so that after the line had been printed, the next instruction executed was *not* the next one in storage, but the one specified by the address part of the Write instruction. This is the simplest example of branching, which is the process of breaking out of the one-after-the-other sequence of storage locations from which instructions are normally executed in the 1401 (and in most machines).

The Write and Branch instruction is an example of an *unconditional branch:* the next instruction is to be taken from the location specified by the address of the branch instruction, regardless of any condition in the machine. This can also be done as a separate operation, not combined with input or output, by using the Branch instruction. The actual and mnemonic operation codes are the same: B. The unconditional Branch instruction has one address, which specifies the location of the next instruction to be executed. This is called the I-address, or instruction address, to emphasize that it refers to an instruction, but it is written in the same position as the A-address.

Branch

FORMAT	Mnemonic B	Op Code B	I-address xxx
FUNCTION	The next instruction is unconditionally taken from the storage location specified by the I-address.		
WORD MARKS	Not affected.		
TIMING	T = .0115 (L_I + 1) ms.		

The more powerful application of the branching idea is in the use of *conditional* branch instructions. With these, the next instruction is taken from the specified address *only if* some condition in the machine is present; otherwise, the next sequential instruction is executed. There are several conditional branch instructions in the 1401. The simplest of them is the *Branch If Indicator On* instruction. Here, the d-character is used to specify what condition in the machine is to be tested, as shown in the summary below.

On a Branch If Indicator On instruction, if the d-character is a blank, the instruction operates as an unconditional Branch. This means, in effect, that if the last instruction of a program is a Branch, with blank storage following, there is no need to put a word mark in the character position immediately following the last instruction. (It *is* necessary to do so otherwise.)

Branch If Indicator On

	Mnemonic	Op Code	I-address	d-character
FORMAT	B	B .	xxx	x

FUNCTION The d-character specifies the indicator tested. If the indicator is on, the next instruction is taken from the location specified by the I-address. If the indicator is off, the next sequential instruction is taken. The valid d-characters and the indicators they test are as follows:

d-character	Branch On:
bl	Unconditional
9	Carriage channel #9
@	Carriage channel #12
A	"Last card" switch (sense switch A)
B	Sense switch B*
C	Sense switch C*
D	Sense switch D*
E	Sense switch E*
F	Sense switch F*
G	Sense switch G*
K	End of reel*†
L	Tape transmission error*
?	Reader error if I/O check stop switch is off †
!	Punch error if I/O check stop switch is off †
P	Printer busy (print storage feature)*
#	Printer error if I/O check stop switch is off †
/	Unequal compare ($B \neq A$)
R	Printer carriage busy (print storage feature)*
S	Equal compare ($B = A$)*
T	Low compare ($B < A$)*
U	High compare ($B > A$)*
Z	Overflow †
%	Processing check with process check switch off †

*Special feature.

†Conditions tested are reset by a Branch If Indicator On instruction.

The indicators tested are not turned off by this instruction except as noted by a †. When carriage tape-channel 9 or 12 is sensed, the corresponding indicator is turned on. These carriage channel-indicators are turned off when any other carriage tape-channel is sensed. The next Compare instruction turns off the compare indicators.

WORD MARKS Not affected.

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

For an example of the use of a conditional Branch, consider the following simple example. We are required to read a deck of less than a hundred cards, print certain items of the information on them, and print the total of one of the fields when the last card has been read. Suppose that the field assignments are as follows:

Card Field	Printing Field
7-13	1-7
4-5	11-12
18-30	16-28
37-40 (Field to be summed)	30-35 sum on line below body of report

The first two printing fields are to be zero-suppressed—that is, any leading zeros are to be omitted in the printing.

This is not so different from examples we have seen before, there being only two new features. The detection of the last card of the deck can be done with a Branch If Indicator On instruction in which the d-character is A, which designates the last card indicator. If sense switch A is on and the last card in the hopper has been read, the branch is taken. If sense switch A is on and cards remain in the hopper, the next sequential instruction is taken. If sense switch A is off and the last card has been read, the machine halts.

The suppression of leading zeros is a matter of ease of use of reports. In most business reports, the meaning of a number like 0008904 is not changed by printing it as 8904, and the report has a neater appearance with the zeros omitted. This applies only to leading zeros; the number should not be printed as 89 4. This suppression of leading zeros is easily accomplished with the Move Characters and Suppress Zeros instruction, which has the actual operation code Z and the mnemonic MCS. The instruction moves characters from the A-field to the B-field, stopping upon detection of a word mark in the A-field. Word marks in the B-field are not inspected, and are in fact erased. Any high-order zeros are then replaced by blanks.

Move Characters and Suppress Zeros

FORMAT	Mnemonic MCS	Op Code Z	A-address xxx	B-address xxx

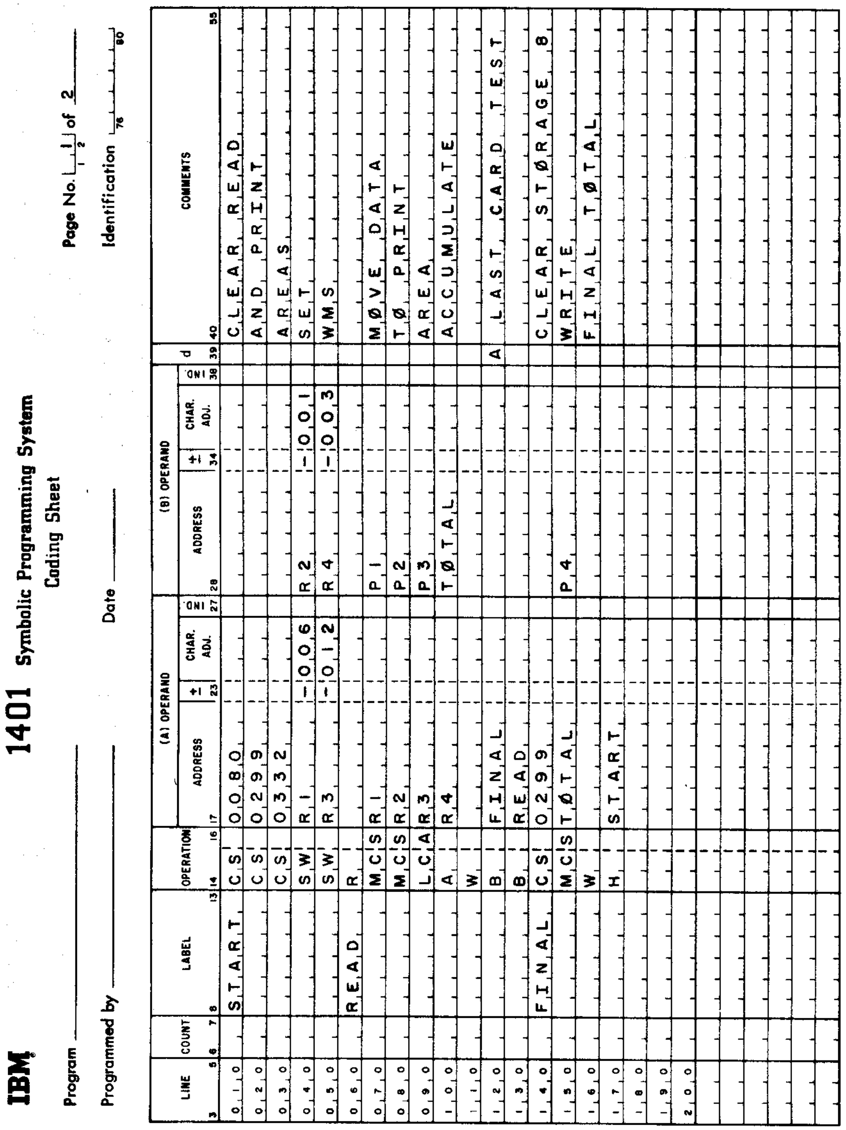
FUNCTION The data in the A-field is moved to the B-field. After the move, high-order zeros are replaced by blanks in the B-field. The sign is removed from the units position of the data field.

WORD MARKS The A-field word mark stops transmission of data. B-field word marks encountered during the move operation are erased.

TIMING $T = .0115\ (L_I + 1 + 3L_A)$ms.

The symbolic program to do this job is shown in Figure 1. As usual, we begin by clearing the read and print storage areas and setting word marks in the read area. The Read a Card instruction is given a label so that it will be possible to refer to it with a later instruction. The numbers for the first two printing fields are moved with a Move Characters and Suppress Zeros instruction, and the third (which was not to be zero-suppressed) is moved with a Load Characters to A Word Mark instruction. Then the card field which is being summed is added into a counter called TOTAL.

This much of the program sets up the printing line and forms the sum. When the line has been printed, the next instruction asks whether the card just read was the last; note the A in the d-character column of the coding sheet. If this was the last card, we branch to the symbolic location FINAL, where there are instructions to print the final total. If this was not the last card, the next sequential instruction is executed, which is also a Branch, but this time an unconditional one which takes us back to read the next card.



IBM 1401 Symbolic Programming System Coding Sheet

Program ______ Programmed by ______ Date ______

Page No. 1 of 2 Identification ______

LINE	COUNT	LABEL	OPERATION	(A) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	(B) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	d	COMMENTS
010		START	CS	0080									CLEAR READ
020			CS	0299									AND PRINT
030			CS	0332									AREAS
040			SW	R1	-	006		R2	-	001			SET
050			SW	R3	-	012		R4	-	003			WMS
060		READ	R										
070			MCS	R1				P1					MØVE DATA
080			MCS	R2				P2					TØ PRINT
090			LCA	R3				P3					AREA
100			A	R4				TØTAL					ACCUMULATE
110			W										
120			B	FINAL								A	LAST CARD TEST
130			B	READ									
140		FINAL	CS	0299									CLEAR STØRAGE 8
150			MCS	TØTAL				P4					WRITE
160			W										FINAL TØTAL
170			H	START									
180													
190													
200													

Figure 1. SPS program illustrating the Branch instruction.

Page No. 2 of 2

LINE	COUNT	LABEL	OPERATION	(A) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	(B) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	d	COMMENTS
010	06	TØTAL	DCW	*		000	0	00					
020		R1	DS	0013									
030		R2	DS	0005									
040		R3	DS	0030									
050		R4	DS	0040									
060		P1	DS	0207									
070		P2	DS	0212									
080		P3	DS	0228									
090		P4	DS	0235									
100			END	START									
110													
120													
130													
140													
150													
160													
170													
180													
190													
200													

Figure 1. Continued

When the last card has been processed, the conditional Branch goes to a Clear Storage instruction to erase the recently printed detail line. The total is moved to the designated print position and the total printed.

The last instruction is a new one, called Halt and Branch. Nothing was said in the problem specification about what should be done once the total is printed. We therefore assume that the machine should be stopped, to wait for another problem to be loaded. The Halt and Branch instruction stops the execution of instructions until the start button on the console is pressed, at which time the next instruction is taken from the location specified by the I-address. In this case, the I-address was made the address of the first instruction of the program. This was done because of the possibility that when one deck of cards had been read, printed, and totaled, it might be desirable to do the same thing with another deck. If this had not been thought necessary, the Halt instruction could have been written without an address, in which case pressing the start button would have caused the next sequential instruction to be executed. In our case the next "instruction" is not an instruction at all, as it happens, but the total that has just been printed. What might happen when the control circuits try to carry out this number as an instruction depends, of course, on what the number is. At any rate, it is not a very desirable situation. It is probably a good practice to put an address on all final halts, to avoid the possibility of this kind of confusion. If there is really nothing more to be done at this point, the I-address of the Halt can be the location of the Halt instruction itself, so that if the start button is pressed the machine will simply halt again.

Halt, Halt and Branch

FORMAT	Mnemonic	Op Code	I-address
	H	.	
	H	.	xxx

FUNCTION The execution of instructions is stopped and the stop-key light on the console is turned on. Pressing the start key causes the program to start at the next sequential instruction if no I-address is written, and to start with the instruction specified by the I-address if one is written.

WORD MARKS No effect.

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

Review Questions

1. *What is a conditional branch instruction?*
2. *Does the Move Characters and Suppress Zeros instruction remove all zeros from the field? Does its action depend on a word mark in the B-field?*
3. *Is the last card switch changed by testing it with a Branch If Indicator On instruction?*
4. *On a Halt and Branch instruction, when does the branch occur?*

5.2 Further Branching Operations

There are a number of other types of branching operations besides those mentioned so far. After mentioning one of these briefly, we shall consider the most important application of the concept: its use in comparison of data fields.

The next instruction to be considered is a rather special one that tests a single character, called Branch If Word Mark and/or Zone. The B-address specifies a character position to be tested. The I-address says where to find the next instruction if the position satisfies the conditions on the word mark and/or zone bits specified by the d-character. The tests are described in the summary below.

This instruction, it may be seen, can test for all combinations of word mark and zone bits. This feature, which is used frequently, saves a great deal of trouble. We shall find several applications for it in later sections.

Branch If Word Mark and/or Zone

FORMAT

Mnemonic	Op Code	I-address
BWZ	V	xxx
B-address	d-character	
xxx	x	

FUNCTION The single character at the B-address is examined for a particular bit configuration, as specified by the d-character. If the bit configuration is present as specified, the program branches to the I-address for the next instruction:

USED often

d-character	Condition
1	Word mark
2	No zone (no-A, no-B-bit)
B	12-zone (AB-bits)
K	11-zone (B, no-A-bit)
S	Zero-zone (A, no-B-bit)
3	Either a word mark, or no zone
C	Either a word mark or 12-zone
L	Either a word mark or 11-zone
T	Either a word mark or zero-zone

WORD MARKS As explained.

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

The most useful application of branching is in combination with the Compare instruction, which lets us compare two fields in storage. The contents of the A and B fields are compared; if they are not the same, the *unequal indicator* is turned on. A Branch If Indicator On instruction can then be used to test this indicator.

The status of the unequal indicator is not affected by testing it with a Branch If Indicator On instruction. Therefore it may be tested several times after being set once, if desired.

Compare

FORMAT

Mnemonic	Op Code	A-address	B-address
C	C	xxx	xxx

FUNCTION The data in the B-field is compared with an equal number of characters in the A-field. The bit configuration of each character in the two fields is compared. The comparison turns on an indicator that can be tested by a subsequent Branch If Indicator On instruction. The indicator is reset by the next compare instruction.

WORD MARKS The first word mark encountered stops the operation. If the A-field is longer than the B-field, extra A-field positions at the left of the B-field word mark are not compared. If the B-field is longer than the A-field, an unequal-compare results.

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

Note: Both fields must have exactly the same bit configurations, to be equal. For example, $00\overset{+}{0}$ compared with $00\bar{0}$ results in an unequal comparison.

As an optional special feature, the 1401 can be equipped with the High-Low-Equal compare device, which considerably expands the power of the Compare instruction. With this device installed, the comparison turns on a separate *equal* indicator if the two fields are equal, and turns on either the *high* or *low* indicator as well as the unequal indicator if they are not the same. "High" and "low" here refer to a scale in which the characters of the machine are ranked from smallest to largest. In this scale, the "smallest" character is a blank, the letters of the alphabet run from A as smallest to Z as largest, and the digits follow the alphabet. The various special characters fit in at the positions shown in Section 12.

When it is necessary to determine which of two signed numerical fields is algebraically larger, it is best to subtract one from the other and use a Branch If Word Mark and/or Zone instruction to determine the sign of the difference. The Compare instruction cannot be used if the fields could have different signs, because it will treat the sign bits as the zone bits of a letter. This is what we want in comparing two alphabetic fields, but not what we want for algebraic comparison.

For a practical illustration of the use of the Compare instruction, we may write the program to perform the first summarization in the sequential file processing example of Section 1.3, with one simplification. It may be recalled that in the example we read the merged master and transaction deck, obtaining the unit price from the master and using it to extend the price of each sale, and summarizing the total sales for each product. This complete job is considered in Exercise 5. Here we shall simplify the task by assuming that the input deck consists only of the extended sales cards—that is, that we are required to summarize the new sales deck. This deck contains a card for each sale, showing the product number, district, salesman, number of units sold, and the total price of the sale. The deck is in product number order. We are required to produce a summary showing the total sales of each product for the month.

A block diagram of the computer processing for this job is shown in Figure 2. We begin with what are called here "housekeeping" operations. These are the preparatory instructions at the start of the program, to clear storage and set word marks. After reading the first card, the part number is moved to the print area and the sales price is moved to a storage field where the total sales for the product will be accumulated. Such a field is often called an *accumulator*. Now another card is read and a comparison used to determine whether it has the same part number. If so, its sales amount is added to the accumulator, a check is made to determine whether this was the last card and, if not, another card is read and the process repeated.

When it is found that a card has a different part number from the previous one, the situation is this: the information from the new card is in the read storage area, the part number of the previous group is in the print area, and the sum of the sales amounts for the previous group is in the SUM. It only remains to edit the total and print the line.

The two last card tests are necessary for the following reasons. It is convenient to use the same editing and printing steps for the last group of cards as are used for all others. This dictates a Branch to the same

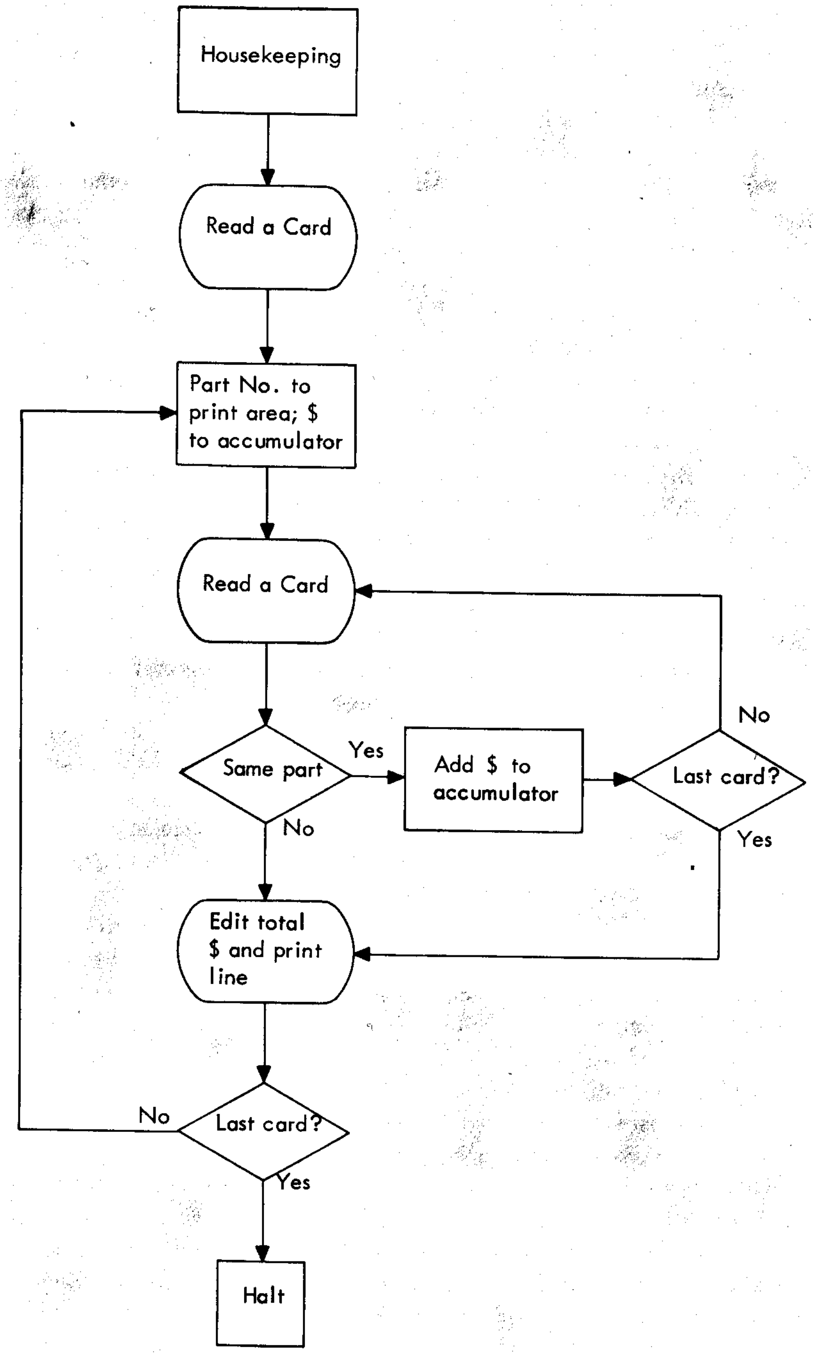


Figure 2. Block diagram of the computer operations in producing the sales summarization by product, in the example of Section 1.3.

steps—after which the computer, of course, has no way of "knowing" that the steps were reached by a different path than normally, and that something different should be done on completing them than is normally done. This is the reason for the second test, after the output box. It is important, in doing this, to know that testing the last card indicator does not turn it off; this is not true of some of the other indicators.

This problem presents an excellent example of a principle that the programmer must never forget: you have to plan for everything. What would happen if the last card of the deck were the only card for a product number? The comparison would show that the previous card was the last of a group, the line for that group would be printed—and the last card test would stop the program without ever processing the last card! (This does not happen if the last card is part of a group of cards having the same product number.) The simplest solution is to put a blank card at the end of the deck, which will take care of the special situation without causing any trouble in the normal case.

(Although this is a simple solution, it is not a particularly desirable situation for the computer operator. Exercise 8 considers a better solution.)

As a general rule, it is an excellent idea to check every block diagram to be sure that such special situations as the first card, the last card, and single-card groups are properly handled. And it is also an excellent idea to be sure that test cases are designed to test such situations. It is most disconcerting to discover after four months of operation that a program does not properly handle some special condition.

With the clear picture of the logic of the program that is provided by a careful study of the block diagram, the program shown in Figure 3 presents no difficulties. The only instruction not previously illustrated is the Compare, which is used here to determine whether the part number in the read area is the same as the part number in the print area. After the comparison, a Branch If Indicator On instruction with a d-character of *slash* is used to test the Unequal Compare indicator. If it is on, the program branches to the edit and print instructions. The Halt instruction is written with an I-address that is the same as its label, so that if the start button is pressed after the program is completed, the Halt will simply be repeated. This prevents an accidental attempt to restart the program when there is nothing more to do. Following the Halt and Branch, there is another Halt. This is provided merely to make sure that there is a word mark in the position following the last instruction to be executed, since every instruction except an unconditional Branch must be followed by a word-marked character.

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	CS	0080									HØUSEKEEPING
020			CS	0299									
030			CS	0332									
040			SW	RPN	−	003		RDØLL	−	005			
050			R										
060		MØVE	LCA	RPN				PPN					PRØDUCT NUMBER
070			MCW	RDØLL				SUM					SALES AMØUNT
080		READ	R										
090			C	RPN				PPN					SAME PR NØ Q
100			B	PRINT								/	PRINT IF NØT
110			A	RDØLL				SUM					ACCUM IF SAME
120			B	PRINT								A	LAST CARD Q
130			B	READ									BACK READ IF NØT
140		PRINT	LCA	EDIT				PDØLL					
150			MCE	SUM				PDØLL					
160			W										
170			B	HALT								A	LAST CARD Q
180			B	MØVE									
190		HALT	H	HALT									
200			H										

Figure 3. SPS program to do the processing defined in the block diagram of Figure 2.

Page No. 2 of 2

LINE	COUNT	LABEL	OPERATION	(A) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	(B) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	d	COMMENTS
010		RPN	DS	0004									
020		RDØLL	DS	0019									
030		PPN	DS	0204									
040		PDØLL	DS	0217									
050	07	SUM	DCW	*									
060	09	EDIT	DCW	*		,		0.					
070			END	START									
080													
090													
100													
110													
120													
130													
140													
150													
160													
170													
180													
190													
200													

Figure 3. Continued

Review Questions

1. *If a second Compare instruction were executed immediately after another Compare, what net effect would the first Compare have on the Unequal Compare indicator?*
2. *Suppose the instruction BWZ 0600 0800 1 is located in 800. What would it do?*
3. *In the program of this section, what happens to the information from the first card of a new group while the line for the previous group is being printed?*

5.3 Case Study: Parts Explosion and Summary

In a manufacturing operation, parts explosion and summary is the process of getting the total parts requirements from a prescribed production schedule of finished goods. In the somewhat simplified example to be considered in this case study, we are given a production schedule deck containing one card for each model to be manufactured, each card showing the quantity of this product required. Each product has its own parts requirements, which are given in a parts requirements deck. This deck contains, for each product the company makes, as many parts cards as there are different parts in the model. Each parts card shows the product number, the part number and description, and the quantity of this part required for the model. The basic task is to find the total number of each part required by the entire production schedule.

The following listing shows in semischematic style the information for two hypothetical models from the catalog of a furniture manufacturer:

Schedule card: model 5392 table; 40 required
Part card: 5392 table requires 1 top, part 278
Part card: 5392 table requires 4 legs, part 339
Part card: 5392 table requires 2 braces, part 447
Part card: 5392 table requires 12 screws, part 2285
Schedule card: model 5673 table; 36 required
Part card: 5673 table requires 1 top, part 276
Part card: 5673 table requires 4 legs, part 339
Part card: 5673 table requires 2 braces, part 447
Part card: 5673 table requires 1 front plate, part 663
Part card: 5673 table requires 18 screws, part 2285

We see that the first model creates a need for:

40 tops, part 278
160 legs, part 339
80 braces, part 447
480 screws, part 2285

The second model creates the need for:

36 tops, part 276
144 legs, part 339
72 braces, part 447
36 front plates, part 663
648 screws, part 2285

The explosion portion of the application produces this type of information, in our example in the form of one card for each type of part required by each model. The summary portion gets the total requirements for each part. In this sample, the summary would show:

36 tops, part 276
40 tops, part 278
304 legs, part 339
152 braces, part 447
36 front plates, part 663
1128 screws, part 2285

This is the general idea of the job. Now we may consider in a little more detail the implementation of the application in terms of the card and report forms to be used here.

A flow chart of the processing is shown in Figure 4. The first step in the job is to obtain the parts requirements of each model and multiply by the number of models to be built. In order to do this, the production schedule is punched into cards having the following format:

Columns 1-5	Model number
Columns 6-9	Number of this model to be built

These cards are next sorted into model-number order, for collating with the master parts requirements file. This file, which is in model number order, consists of cards having the following format:

Columns 1-5	Model number
Columns 6-10	Part number
Columns 11-30	Part description
Columns 31-33	Number of this part required for one of this model

This deck will contain, for each model, as many cards as there are component parts in the model.

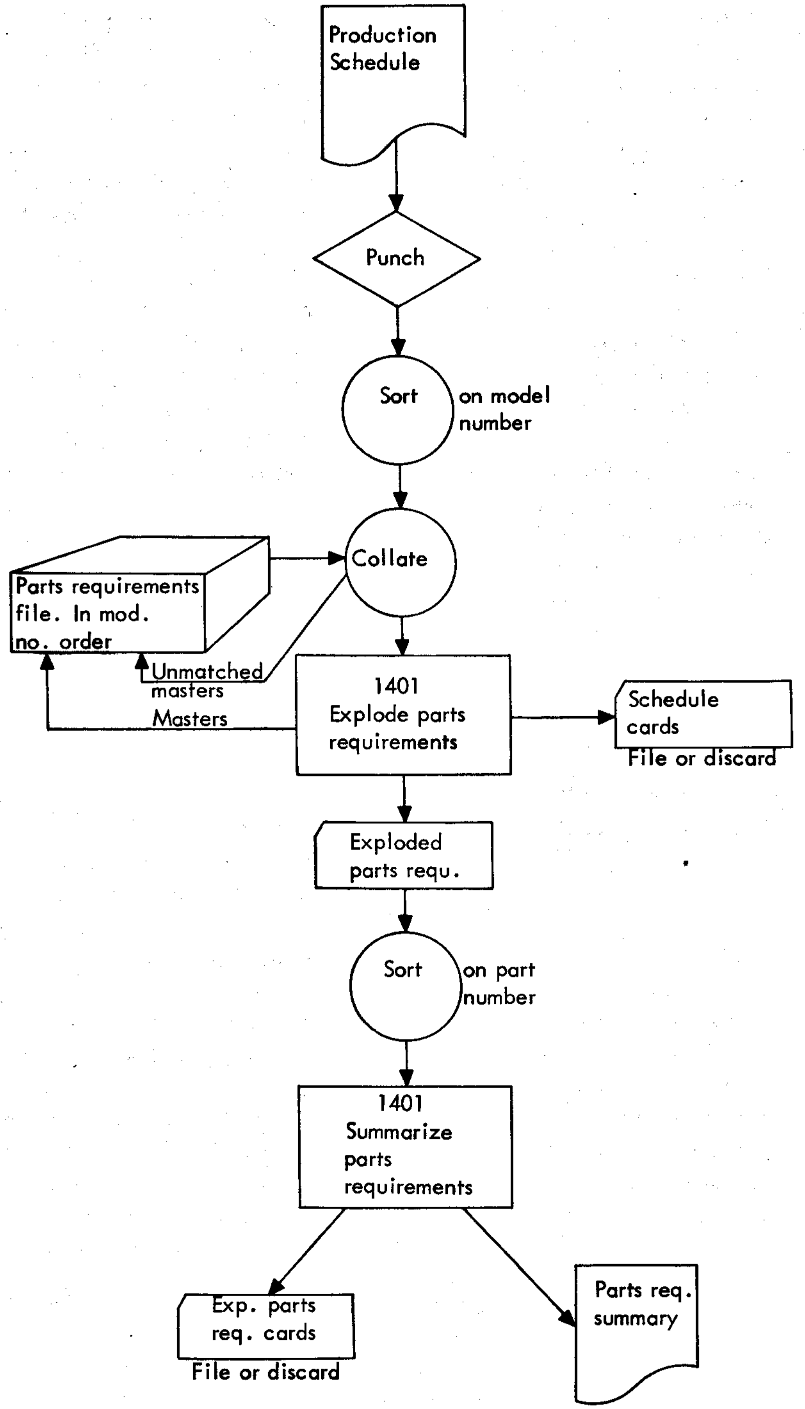


Figure 4. Flow chart of a procedure for parts explosion and summary.

Not every model in the catalog will be built in any one production period, ordinarily, so when the parts requirements master file is collated with the schedule cards, there will be unmatched masters. These could be left in the deck, but it will simplify our block diagramming and coding work here if we assume that they are selected out of the deck and returned to the file. In fact, it might work out in practice that the unmatched masters would make a much larger deck than the matched, so that it would be entirely reasonable to remove the unneeded ones to avoid wasting the computer time required to read them.

The deck that now goes to the computer consists of sets of what may be called "packets," each packet containing a schedule card giving the number of a certain model to be built, followed by parts requirements cards showing what parts are required to build one of the model and how many of each. The task of the computer run is to "explode" the parts needed for each model—that is, multiply the number of each model to be built, by the quantity of each of the various parts which are used to build it. This will produce another deck of cards, each card giving the quantity of some part needed to build the specified number of units of one model.

After these cards are sorted on part number, a second computer run can easily summarize the number of each part needed by the various models in which it is used.

The format of these cards is:

Columns 1-5	Model number
Columns 6-10	Part number
Columns 11-30	Part description
Columns 31-35	Quantity of this part required to build the specified number of this model

The format of the parts requirement summary is:

Positions 1-5	Part number
Positions 10-16	Quantity required

A block diagram of the computer processing to explode the parts requirements of each model in the production schedule is shown in Figure 5, and the symbolic program in Figure 6. The housekeeping is much the same as before, except that now we clear the punch area instead of the print area. The setting of word marks is done in absolute, as a concession to the necessity of taking advantage of the similarity of card formats in order to avoid setting word marks separately for each of the two different types of cards that must be read.

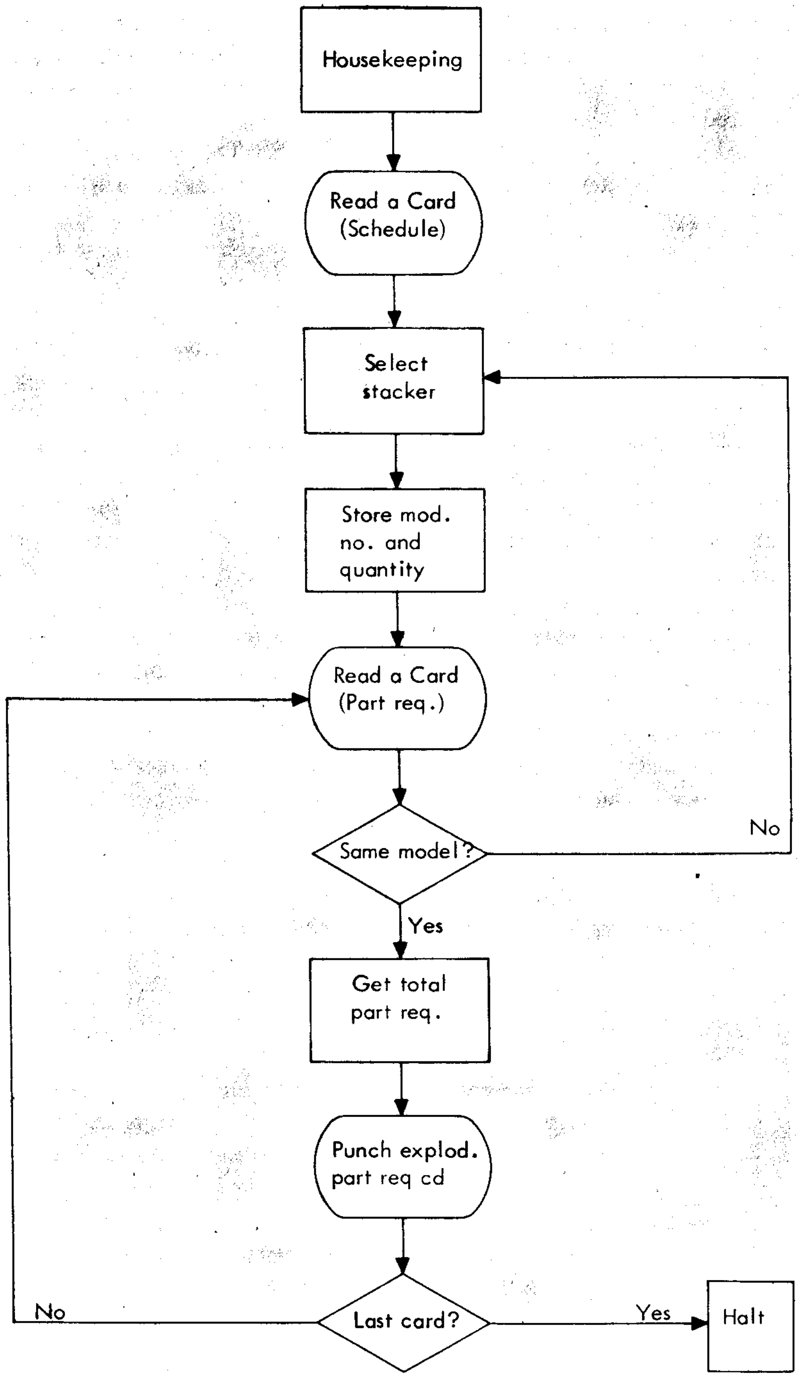


Figure 5. Block diagram of the explosion part of the procedure charted in Figure 4.

The first card in the deck should be a schedule card. After reading it, we use stacker selection to put this card in the 1 pocket, instead of the normal read pocket in which the parts requirement cards will be stacked. Stacker selection requires the Select Stacker instruction. The actual operation code is K and the mnemonic SS. The instruction needs only a d-character besides the operation code, to determine which of the stackers is to be selected.

Next, the model number is placed in the punch storage area from which it will be punched, and where it may be used to compare with the model number from succeeding cards to determine when a new model is to be processed. The model quantity is next moved to a working storage location named QTY, for later use in multiplying by the number of each part required.

Now another card is read, which should be a parts requirement card this time. (There obviously must be at least one part to each model.) As later cards are read, however, we shall eventually come to the model requirement card for the next model. Therefore, the first thing to do now is to determine whether the model number on this card is the same as that on the previous card. If it is the same, we set up the information for the total part requirement card, multiply the number of models by the quantity of this part required for one model, and punch the card. If the comparison showed a different model number, then the previous model must have been completely processed, this must be a new schedule card, and we go back to select the stacker and proceed with the processing of the new model.

After punching the card, a last-card test is used to determine whether the end of the deck has been reached. If it has, we halt; if not, we return to read another card.

The summarization run is not hard to write, and is left as an exercise.

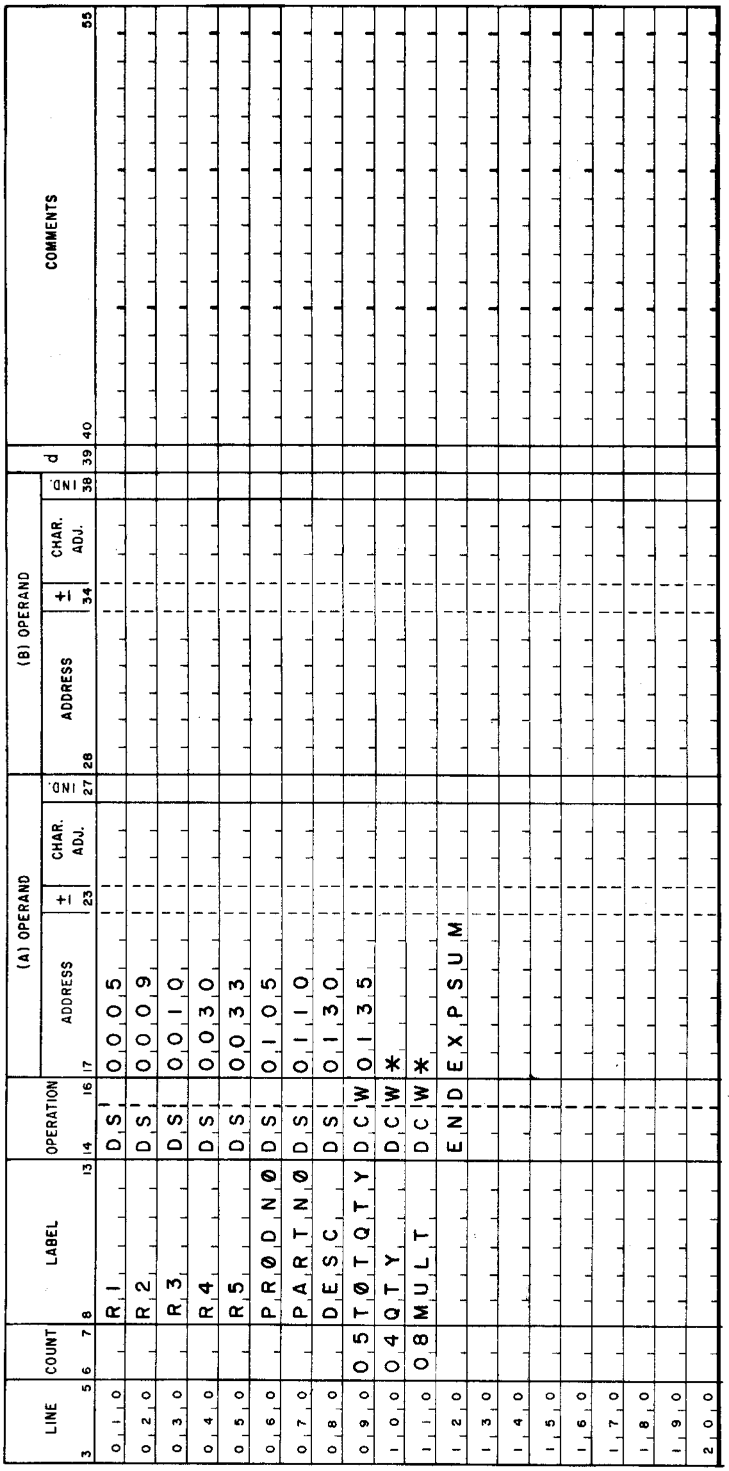
Review Questions

1. *It is rather essential to the procedure of the block diagram that there not be two schedule cards for the same model. What would happen if there were? Would it make any difference whether this happened at the front of the deck instead of the middle or the end?*
2. *What would happen if there were a schedule card for which there were no corresponding parts requirements cards? Would it make any difference where in the deck this happened?*
3. *This procedure contains no error checking of any kind. Can you think of a way to use a total count of all parts in all models to provide some measure of checking of card handling?*

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		EXPSUM	CS	0080									HØUSEKEEPING
020			CS	0180									
030			SW	0001				0006					
040			SW	0011				0031					
050			R										READ SCHED CARD
060		SCHCD	SS									1	SELECT STACKER
070			LCA	R1				PRØDNØ					PRØDUCT NJMBER
080			MCW	R2				QTY					QUANTITY SCHED
090		PARTCD	R										
100			C	R1				PRØDNØ					SAME PRØD NØ Q
110			B	SCHCD								/	NØ-NEW SCH CARD
120			LCA	R3				PARTNØ					YES-PART CARD
130			LCA	R4				DESC					SETUP PUNCH AREA
140			MCW	QTY				MULT	−	004			GET
150			M	R5				MULT					TØTAL
160			MCW	MULT				TØTQTY					QUANTITY
170			P										PUNCH EXP PT REQ
180			B	HALT								A	LAST CARD Q
190			B	PARTCD									NØ-READ NEXT CD
200		HALT	H	HALT									

Figure 6. Program of the computer operations diagrammed in Figure 5.



Page No. 2 of 2

LINE	COUNT	LABEL	OPERATION	(A) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	(B) OPERAND ADDRESS	±	CHAR. ADJ.	IND.	d	COMMENTS
010		R1	DS	0005									
020		R2	DS	0009									
030		R3	DS	0010									
040		R4	DS	0030									
050		R5	DS	0033									
060		PRØDNØ	DS	0105									
070		PARTNØ	DS	0110									
080		DESC	DS	0130									
090	05	TØTQTY	DCW	0135									
100	04	QTY	DCW	*									
110	08	MULT	DCW	*									
120			END	EXPSUM									
130													
140													
150													
160													
170													
180													
190													
200													

Figure 6. Continued

Exercises

*1. Using the High-Low-Equal compare feature, move whichever of the two fields DATA1 and DATA2 is *larger*, to location BIG. (Do not write a complete program. That is, assume word marks are set and that the symbols will be defined elsewhere.) Draw a block diagram and write a program segment.

2. Move whichever of the three fields DATA1, DATA2 and DATA3 is *largest*, to location BIG. (Assume all the words are different.) Draw a block diagram and write a program segment with same assumptions as in Exercise 1. *Hint:* Place the larger of DATA1 and DATA2 in BIG, then compare this number with DATA3 and replace it with DATA3 if DATA3 is larger.

*3. Read a card. If column 23 contains an 11-zone punch—regardless of what else the column contains—*punch* another card containing in columns 1-40 the information in columns 41-80 of this card. If column 23 does not have an 11-zone punch, *print* in positions 1-40 the information in columns 41-80. Write a program segment, in absolute if desired.

4. In the block diagram and program of Section 5.2, add the steps necessary to produce a sales total for the month, for all products.

5. Draw a block diagram and write a complete program to do the extension and first summarization of the example in Section 1.3.

The format of the master cards is:

Columns 1-4	Product number
Columns 5-9	Unit price
Column 10	11-zone

The format of the sales cards is:

Columns 1-4	Product number
Columns 5-8	Units sold
Columns 9-11	Salesman
Columns 12-13	District

See Exercise 4 of Section 1 for a description of the processing and some hints on how to proceed.

*6. Suppose that cost information is included in the parts cards in the case study of Section 5.3, as follows. Columns 34-38 contain the total cost of as many of this part as are required to build one of this model. Modify the block diagram and program as necessary to provide a parts cost total for each model scheduled for production. This should be printed during the explosion run, in the following format:

Position 1-5	Model number
Position 10-13	Number to be built
Position 18-25	Total cost of all parts required to build this many of this model. Print with decimal point.

7. After the exploded parts requirements cards have been sorted into part number order, they must be summarized by part number. Draw a block diagram and write a complete program.

8. Extend the block diagram of Figure 2 to handle the special case of a one-card "group" at the end of the deck, without the requirement of a blank card at the end.