The RCA 501 Assembly System

H. BROMBERG†, T. M. HUREWITZ†, AND K. KOZARSKY†

CURRENT techniques in automatic coding attempt to shift the user's tasks away from the computer and closer to his application. Sacrificing coding details in this way, it is believed that monumental savings will result both in computer acceptability and utilization since everyone is now able to describe his own programs. Universal acceptance of problem oriented languages has, for several reasons, not yet followed. One influence is that the generated object programs reflect the adroitness of the executive routine and the remoteness of the input language.

In a recent count, there appears to be well over one hundred automatic coding systems produced for twenty or more different computers. This reflects the recognition of the disparity that exists between the methods of problem preparation and actual problem solution.

By most methods of classification, these hundred-odd automatic codes range more or less continuously between extremes. They vary considerably in complexity, extent of problem area of useful application, and in range of intended user.

One categorization used to differentiate these automatic codes is by the sophistication of the input language—particularly whether this language is "problem oriented" or "machine oriented." However, it must be admitted that if a ranking of these automatic codes is made according to efficiency of the object program, the list would tend to be in nearly inverse order to that obtained by ordering on the level of the input language. It should also be noted that evaluations of the academic aspects of these automatic codes are often greatly at variance with the judgments of the occasionally unfortunate users of these routines.

This is not to say that object program efficiency is the only value criterion of an automatic system. Frequently for short programs or where the capacity of the data-processing equipment greatly exceeds the required performance, it is almost irrelevant. But, in instances where object program efficiency is significant, alternative coding procedures are desirable.

It is conceded that the Problem Oriented Language deservedly has greater prestige than the Machine Oriented Language and greater theoretical interest (at least from a philosophic or linguistic point of view). Nevertheless, the current mechanization of these languages and the distribution of computer expenses dictate demands for both types. It is recognized that direct, facile communication between the layman and his computer as well as the advantage of interhuman communication of the problem definition are obtainable from a Problem Oriented Language. However, there are also needs for programs handling tasks near the limits of the equipments' capabilities as well as for infrequently changing, very highly repetitive, data-processing routines.

One of the often expressed goals in automatic coding is the development of complete problem oriented languages entirely independent of any computer. To produce any "most efficient" coding in this circumstance means that, among other things, psychological inferences as to the intentions of the writer are to be made by the automatic code. Furthermore, the apparent trend in machine design toward many simultaneous asynchronous operations, multiprogramming and the like, increase the problems associated with producing efficient machine programs from a problem oriented language. It is hardly unreasonable for a user of the new potentially powerful systems to request a coding scheme capable of using these complex, expensive features.

It appears likely then, in the near future at least, that some problem oriented languages will be augmented by some prosaic statements, directly or indirectly computer-related, which will permit attainment of a more "most efficient" machine code. Similarly, machine oriented languages may also yield to this trend and incorporate some features, within their inherent limitations, which tend to be associated with problem language codes.

All this may justifiably be construed as motivation for the automatic routines offered with the RCA 501. The first of these, a machine oriented automatic code, the RCA 501 Automatic Assembly System, is described in this paper.

The Assembly System provides for: relative addressing of instructions and data; symbolic references for constants and data; macro-instructions and subroutines; variable addresses; and descriptor verbs.

Some Machine Characteristics

It is appropriate, as background for what follows, to describe briefly some of those features of the 501 Computer (Fig. 1) which have influenced the design of the Automatic Assembly System. (Incidentally, this computer has been in operation in Camden since April, 1958.)

The 501 Computer has a magnetic core storage with a capacity of 16,000 characters, which is increasable in steps of the same to a maximum of 262,000 characters. Each character, consisting of six information bits and one parity bit, is addressable, although four characters are retrieved in a single memory access. Binary addressing of the memory is provided, requiring 18 bits or three characters per address.

† RCA, Camden, N. J.

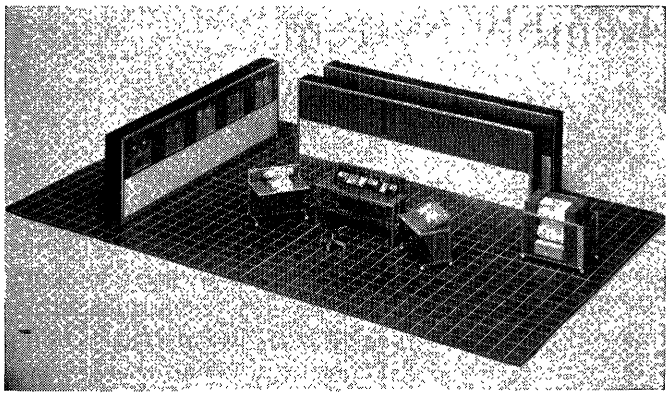
Fig. 1—The 501 computer.

Operation Code	*A* Address	Address Modifier	*B* Address
O	*AAA*	*N*	*BBB*

Fig. 2—A description of one instruction.

P	*A* & *B*	*T*
Stores Address of Next Instruction	Store Address of Operands During Execution	Third Memory Address

Fig. 3—Control registers.

Start Message and End Message	Item Separator
(⟨) (⟩)	(·)
Define Message on Tape and Control Operations in Memory	Controls Operations in Memory

Typical Message ⟨·12564·John-Doe·8934·7⟩

Fig. 4—Control symbols.

The instruction complement consists of 49 two-address instructions (see Fig. 2). Each instruction consists of:

One character for the operation symbol
Three characters for the *A* address
One character for the selection of address modifiers
Three characters for the *B* address,

for a total of eight characters or sixteen octal digits.

There are several control registers, each of which stores 3 characters, or one address (see Fig. 3). The *A* and *B* registers are used to store the *A* and *B* addresses of operands during their execution. The *P* or program register stores the memory address of the next instruction in sequence. The *T* register stores a memory address which is made use of by certain instructions which require three addresses during their execution. All of these registers are addressable and therefore directly accessible to the program.

Seven address modifiers are available. Four of these are standard memory locations and three are the *A*, *T*, and *P* registers. Use of the *P* register as an address modifier permits the writing of self-relative machine coding which may be operated, without modification, in any part of the memory.

The RCA 501 exploits certain control symbols (Fig. 4) in the data. The Start and End Message control symbols define a message on tape, and in the memory also act as control symbols for certain instructions. The Item Separator control symbol is not used as such on tape, but is used in the memory to control certain operations. These control symbols permit variable item lengths and variable message lengths both on tape and in the memory. The entire message may be variable, dependent upon the number and size of the individual items. The instruction complement includes both symbol controlled and address controlled operations.

The 501 includes provision for simultaneous read-write, read-compute and write-compute. This is accomplished by designating magnetic tape instructions as "potentially simultaneous" and establishing a program controlled gate between the normal and simultaneous modes of operation. Thus, the programmer can permit completely automatic switching of tape instructions to the simultaneous mode or he may optionally bracket off portions of the program where such switching is inconvenient.

As for the 501 Assembly System, there are two programmer-prepared inputs. To guide the Assembly System in generating a running program from the pseudocode, the user provides the system with a description of the data files which the program is designed to process. A portion of a data sheet is illustrated in Fig. 5.

Certain auxiliary computations are performed on these data sheets and the results printed out for the programmer's information—such as average message lengths, approximate tape passage time, and weighted average.

Data Addressing

Completely variable length data, on the one hand, yields economies in tape storage and effective file passage time; on the other hand, it presents certain problems with respect to symbolically addressing data items in the memory. These problems are handled in several different ways:

1) Those items whose lengths are fixed relative to the beginning of the message may, of course, be directly addressed by the data name designated in the data sheets.
2) The variable length items of a message may be transferred to a working storage area of memory where space is allocated for the maximum possible size of each variable length item. A single pseudo-instruction performs this function. From this point on, the variable length item may be directly

Item No.	Sub Item	Abbreviation	Description	FAA	JY	Sign	No. Char.		% Use	Wtd. Avg.
							Max.	Avg.		
1		Date	File Label	X	R		6	6	100	
	A	Month					2	2		
	B	Day					2	2		
	C	Year					2	2		

Fig. 5—The automatic code data sheet.

Instr. No.	Comments	OP	*A* Address	*B* Address	*T* Address	C S G	IF	GO TO	IF	GO TO	IF	GO TO

Fig. 6—The automatic code program sheet.

Instr. No.	Comments	OP	*A* Adress	*B* Address	*T* Address	C S G	IF	GO TO	IF	GO TO	IF	GO TO
PRO 1		LRF	POLCY									
	(Macro-Instruction)	TEST	POLCY				EF	PRO40+1	ED	PDQ8+10		
		SC	DATE	DATE	"122558"		+	PRO23+5	−	PHI 15	O	N
PRO 20	"Extra Beneficiary"	DEFK	KBEN									
		DA	RATE	TAX	WRATE							
		ADV		+V4								

Fig. 7—Assembly pseudocode entries.

addressed by the data name preceded with a *W* (representing working storage).

3) It is possible to locate the address of any item in a message by using an instruction which scans a message searching for and counting the control symbols defining items. This instruction leaves the address of the item in an address modifier.

The second input, whose format is shown in Fig. 6, is the pseudocode written on the program sheets.

This is seen to be an expansion of the machine code format which normally includes the operation field and the two addresses *A* and *B*. This is augmented on this sheet by the *T* or third address which for several machine instructions requires presetting. Furthermore, there are provisions for 3 "IF-GO TO" statements providing for conditional or unconditional transfers of control.

The inclusion of these IF-GO TO statements as an optional part of every pseudoinstruction line has two primary motivations. First, it accommodates as a single pseudocode statement the function "Compare and Jump" which has a relatively high frequency in data-processing problems. Second, about $\frac{1}{3}$ of the 501 instructions automatically set a register to 1 of 3 states depending on conditions encountered during the operation of the instructions. Branching instructions may then be used to select different paths depending on the setting of the 3 state register and are easily designated in the IF-GO TO columns.

A single character entry in the CSG column generates an instruction to open or close the simultaneous gate, controlling the phasing of simultaneous tape operations.

Variable Instruction Generation

An interesting feature of the Assembler is the handling of the normal complement of 501 machine instructions. Each of these instructions, when used in the expanded pseudocode format, assumes the identity of a macro-instruction. Up to five, two address machine instructions may be generated by employing a single machine operation code with appropriate entries along the pseudocode line. As an example, the 501 instruction, Decimal Substract, may accomplish not only a three address subtraction, but also a simultaneous gate operation, and transfers of control dependent upon the sign of the difference.

Symbolic and Relative Addressing

Included in the automatic system are such features as mnemonic operation codes, symbolic and relative addressing including the use of both alphanumeric and octal literals for constants, and acceptability of machine code should it be desired. Fig. 7 shows examples of pseudocode entries.

The Instruction number is composed of characters designating the page and line number of the instruction.

When addressing an instruction, reference is made to that instruction whose elements appear in the Operation, *A*, and *B* fields of the program sheet. Since, however, one single pseudoinstruction may account for as many as five machine instructions, it is desirable to address those. Therefore, a stipulated suffix to an instruction number will allow a reference to any generated instruction and to any field or desired character within one of these instructions.

Relative addressing of these symbolics enables the programmer to refer to the *N*th pseudoinstruction following or preceding a given symbolic. The program will be ordered by page number in alphabetic sequence and within pages by line number before any processing is undertaken. Accordingly, to accomplish an insertion, one need only assign appropriately sequential labels to the desired instructions and the program will place them in the proper positions.

It is not necessary, however, to label every instruction. Relative addressing allows reference to be made to unlabeled instructions in the program. One might usually expect to be labeled the first of a sequence of instructions performing a logical function and those to which frequent reference is made by other instructions, but this is left solely to the discretion of the programmer.

Descriptor Verbs

Descriptor verbs constitute an important part of the automatic code. These verbs contribute, in general, only to the description of the program and do not become a directly converted active part of the machine code.

These special verbs perform a variety of functions such as the definition of program segments, overlaying memory regions, reserving areas of memory, extracting the machine address corresponding to any symbolic name, defining constants and variables and providing for insertions, deletions, and corrections in pseudocode. These verbs are executed during assembly and are deleted from the final program.

Variable Addressing

Another programming aid incorporated within the system is the variable address feature. A variable address allows the specification of addresses or constants to be symbolically named and to be defined later in the program. A variable may be substituted for any other machine or symbolic address in any instruction. This feature, for example, permits tagging, as a variable, the address of an instruction not yet written. It is only necessary then, at a subsequently convenient moment, to employ the Descriptor Verb, "Define V," to supply the actual address of this variable for every place it was used.

It is also possible to use variable addresses in addition to any machine or symbolic address. This is accomplished by placing the variable address in the same column as the one to which the variable is to be applied and in the directly succeeding line. A plus or minus prefix will then specify addition or subtraction of the variable address. A variable to be added or subtracted will not be applied until the variable is converted to an actual machine address. The use of variable addresses, then, allows for symbolically designated modification of the program at the actual or machine code level.

Literals

Literals, or constants whose address and name are identical, are used in the assembler. Two types of literals are provided, alphanumeric literals for operations with data, and octal literals for operation with instructions which, it has been noted, are binary coded.

A literal is normally carried along with the segment in which it appears. However, a terminal character of the literal may be used to specify that the literal be stored in a common constant pool available to all segments of a program. A terminal character may also be used to designate and to differentiate among duplicated copies of the same literal in the program. Here, too, these duplicate literals may be associated with the segment or with the common pool of constants.

Alternatively, of course, constants may be defined by a "Define Constant" Descriptor Verb and assigned an arbitrary symbolic address. Terminal characters on these constants perform the same functions with these constants as those just described.

Macro-Instructions

The macro-instructions included with the Assembler create 2 address symbolic coding which is spliced directly into the main body of coding in place of the macro-instruction pseudocode call-line. A single macro-instruction will generate all of the instructions required to perform some task which would normally require the writing of a sequence of machine instructions.

Parameters, which the macro-instruction uses, are specified at the pseudocode call-line by the programmer. No restrictions exist as to number or size of these parameters. If a macro-instruction is to be generative, it contains one other part aside from the main body of stored coding. This part decides, from an interrogation of call-line parameters, which particular set of macro-instruction coding is to be included in the main routine.

Subroutines

The assembly system provides for an expandable library of subroutines to be available to the programmer. These subroutines generate assembly language pseudocode and as such may use all the assembly features such as macro-instructions, descriptor verbs, and so forth. Subroutines may be open or closed and generative or fixed.

Parameters for subroutines are specified at the pseudocode calling line. For open subroutines, parameters are incorporated during the operation of the assembler. These parameters may merely be substituted in the subroutine as in the case of a fixed routine or may be subject to considerable testing and manipulation as occurs with a generative subroutine.

Closed subroutines may either incorporate parameters during assembly or use parameters generated by the running machine program. In this case the parameters are located relative to the subroutine call-line.

The design of the system is open to the extent that any useful number of macro-instructions and subroutines may be added.

Provisions for Program Modification

The Assembly System offers two main listings for program up-dating. First, listings are given of the object machine code and the Assembly language pseudocode. Second, is a list of all symbolic addresses and those instructions referring to them. In addition, the Assembly System generates an information block preceding each object program. This block, which contains all program stops, breakpoint switches, and tape addresses is available for input to a service routine which will modify any corresponding entries within the object program.

There are two types of error indicators used by the Assembler. One causes the Assembly System to print the source of trouble and stop immediately. The other and major class consists of on-line printed statements indicating the type and location of errors. In this case the Assembly System continues its functions ignoring the "guilty" statements until all such indicators have been found. This permits the user to specify corrective measures for all errors at one time.

In summary then, the 501 Assembly System lies in an intermediate category. On the one hand, it is definitely machine oriented, amplifying the 501 instruction complement and requiring a knowledge of the 501. However, it also provides for a flexibility of order statements, not confined to the 2 address machine order code. A variable number of machine instructions are generated dependent upon the number and types of entries made on each pseudocode line. Both macro-instructions and subroutines may be of the generative type and since the library is open-ended, may be augmented whenever necessary.

In short, the RCA 501 Assembly System is a programmer's aide, enabling him to make maximum use of machine capabilities with a minimum of clerical effort.

Acknowledgment

The authors acknowledge the extensive contributions of M. J. Sendrow, who participated in the planning and creation of the RCA 501 Assembly System.

A Program to Draw Multilevel Flow Charts

LOIS M. HAIBT[†]

Introduction

THE preparation of a program for a digital computer is not complete when a list of instructions has been written. It still must be determined that the instructions do the required job, and if necessary the instructions must be changed until they do. Also a description of the program should be written for others who may want to understand the program. A useful tool for the last purpose is a graphical outline of the program—a flow chart.

Flow charts serve two important purposes: making a program clear to someone who wishes to know about it, and aiding the programmer himself to check that the program as written does the required job. A flow chart drawn by the programmer would serve for the first purpose, but drawing one is often a tedious job which may or may not be done well. For the second purpose, it is important to have the flow charts show accurately what the program does rather than what the programmer might expect it to do. Consequently, it was decided to write a program, the Flowcharter, for the IBM 704 to produce flow charts automatically from a list of in-

[†] IBM Res. Center, Yorktown Heights, N. Y.