of subroutines permits the coder to think in terms of functions that are complex combinations of the elementary arithmetic and logical operations of the machine. This is in effect a different structure than that permitted by the basic machine. The translation problem in this case is the replacement of the coder's expression for the complex operation by the equivalent structure in the machine terminology. A system is now under development at the Naval Research Laboratory to perform this translation for a single-address computer, the NAREC. For purely arithmetic operations this system will permit the coder to specify in each instruction, written using a redundant dictionary, up to six arguments for a subroutine, designation of the subroutine, where to store the results of the subroutine and an arbitrary transfer of control if necessary. The translation process uses only an IBM sorter and reproducer and takes place in two steps: (1) translation of the structure, and (2) translation of the words. In the first step the deck containing the instructions is sorted by subroutines and the number of occurrences of each subroutine is counted. Standard sets of cards are prepared for each subroutine. These contain the complete coded pattern of the subroutines with constants in redundant form and with blanks left for the positions occupied by the arguments. In addition, the subroutine decks contain punches that control the read feed of the reproducer and the timing of selectors that punch the arguments into the appropriate positions in the subroutine deck. The instruction deck is placed in the read feed, the subroutine decks in the punch feed. As each instruction card passes under the reading brushes the information is punched into the first card of the subroutine deck. This information is gang-punched back into successive cards of the subroutine deck while the read feed is held. Selectors then transfer the symbols from the gang-punched columns into appropriate positions on the subroutine cards. When a subroutine is completed the next instruction card is read. This process results in a deck containing a set of instructions in terms of the elementary operations of the computer but with symbols still from a redundant dictionary. The second step of word translation is performed as described previously.

The computer itself can obviously perform the processes described above. Only a careful analysis can determine whether or not any advantage would be gained by using the computer for this purpose instead of the elementary machinery now in use.

COMPUTER AIDS TO CODE CHECKING

By

Ira C. Diehm
National Bureau of Standards, Washington

When a complex routine is tried on a computer for the first time, it is seldom found to be free from error. The trend toward automatic performance of the clerical parts of the coding process should reduce the number of coding errors. This mechanization of coding is the subject of several papers at this conference. Nevertheless, a significant amount of valuable computer time will continue to be devoted to the search for coding errors.

Careful proofreading and clerical checking are obvious but important

methods of eliminating errors before going to the computer. On the other hand, our machines are intended to help to eliminate such drudgeries, so that we are interested in how the machines themselves can be used to analyze coding errors.

In the first place, most computers have convenient built-in features which are useful to this end. The term "breakpoint" is used to mean a special programmed halt which may be overridden by a manual switch. Many machines have breakpoint provisions, means for determining the contents of particular memory locations, and means for determining what instruction is being executed. In addition to these, SEAC, the National Bureau of Standards Eastern Automatic computer, for example, has a device called the "automonitor", which will automatically record each instruction and its result. This device was not originally on the machine. One of the first diagnostic routines we used was a routine of the interpretive type which recorded each instruction and its result, while the routine was apparently being executed in the normal manner. The engineers noted that this could be done automatically with a very few engineering changes, so they added the "automonitor." The device can also be used to record the instruction and result only on predetermined instructions, in fact it is most valuable when used in this way.

Such built-in features are convenient, but in their absence one can program equivalent measures or better ones, on any adequate general purpose computer. It is these auxiliary routines, and the philosophy embodied in them, that chiefly determine the efficiency with which the machine is used in finding coding errors.

The important principles to be followed, I believe, are that the procedures to be used at the computer should be planned in advance and mechanized as much as possible. There should be no attempt at human analysis of errors while at the computer. It is my feeling that the computer should not be operated by the programmer, for he will not follow a predetermined plan, but will make on-the-spot improvisations which are usually regretted later, and will leave the machine idle while he speculates on possible causes of errors. The operation is best done by another person, preferably one who devotes most of his time to computer operation and is, therefore, completely familiar with the controls and has an intuitive feeling, gained from experience, of possible machine errors.

Operations which can be done automatically by auxiliary routines should be done automatically rather than by manual operation of the controls; first, because operator errors are very common, and second because the computer could do hundreds of operations while a human being is reaching for a switch. Those operations which must be done manually should be specified completely by means of written step-by-step instructions to the operator.

In fact these principles should apply to computer programming and operation in general, and not just to code checking.

For most efficient use of both computer time and preparation time, the session at the computer should be short. On SEAC, a few hours each week are devoted exclusively to code checking periods of roughly ten minutes each, one routine after another being put on the computer for testing. One can usually get enough information to isolate one coding error in a ten minute period.

Sometimes, however, one wishes to get information about as many different errors as possible in a single run at the computer. A code checking system suitable for this purpose has been designed by J. H. Wegstein of the NBS Computation Laboratory. In this method a sequence of cells called a "control tank" is set aside, and at each of several selected key points in the routine, the control is sent to different cells in this tank. During normal operation each of these cells will simply refer control to the next part of the routine, without breaking the continuity of the calculation, but during code checking special orders which print selected quantities are put into these cells. If these quantities are incorrect, the computer is halted, and previously prepared correct values are inserted before continuing.

Quite a few auxiliary routines have been tried out on various machines. There are a great many possibilities, but all the ideas involved are simple and would occur to nearly anyone who devotes a good deal of time to the use of a machine. The trick is to find useful combinations of ideas.

Some simple routines which have turned out to be used on SEAC are:

1. A routine which makes it easy to read out any specified sequence of cells.

2. A routine which automatically inserts or removes breakpoints in specified locations.

3. A routine which transfers the contents of the memory to magnetic wire or tape. This is used to avoid repetition of correct computations while correcting later errors in a routine. It is also useful in bridging interruptions in normal operation, and may be used, periodically, as a precaution against computer errors.

4. A routine which compares the information on the input medium with the information in the memory, and reads out the addresses and contents of cells where discrepancies occur. This routine has sometimes shown up computer errors as well. The programmers for the Whirlwind have also noted that such a routine was valuable.

Several more involved routines have been useful in finding well hidden coding errors. These include:

1. An interpretive routine which provides a complete history of a specified cell. Each time an instruction affects that cell, the address of the instruction and its result are read out. This routine is usually applied to a location where an

incorrect quantity is known to appear, to a cell whose contents have changed for an undetermined reason, or to an instruction containing variable addresses; but also has incidental uses, such as looking at successive terms of a series without altering the coding.

2. An interpretive routine which determines the path which the control has taken through another routine. At each comparison order, this routine reads out the address of that order, and the address from which the next order will be taken. This routine serves much the same purpose as the EDSAC routine which reads out the operation symbol of each instruction executed.

3. A routine which determines the total effect, on the high speed memory, of each of several chosen sections of the routine being tested. Check points in this latter routine are picked by the programmer, and specified to the computer. The effect of the operation is that the computer executes one section of the routine being tested, then reads out the addresses and contents of memory locations which have been altered by that section. Then it executes another section and reads out the changes it has made, and so on. The computer accomplishes this by first duplicating the memory contents on a magnetic tape and replacing the instructions at check points by instructions which will transfer control to the auxiliary routine. Arrangement is made, of course, for the execution of the replaced instructions at the proper time. Then the first section of the main routine is entered, not in the interpretive manner, but directly. When a check point is reached, the information in the memory and on the magnetic tape are compared, and the addresses and new contents of locations which have been changed are read out. Then the present contents of the memory are duplicated on magnetic tapes and control returns to the routine being tested until another check point is reached.

Effectively, this auxiliary routine uses only the first 8 memory locations, conventionally reserved, on SEAC, for the initial read-in instructions. Actually it temporarily uses other parts of the memory by first transferring their contents to magnetic tapes and later restoring them.

What one tries to achieve in designing such auxiliary routines is to program the machine to select the pertinent information rather than to read out large quantites of data which must be searched through by a programmer.

I wish to acknowledge the contributions of C.J. Swift and J.H. Wegstein of the NBS Computation Laboratory to the ideas presented here.

Note on 'Computer Aids to Code Checking'
by I.C. Diehm, N.B.S.

Programmers of the Eckert-Mauchly Division are using a routine that has proved helpful in detecting one class of programming errors -- errors in the address portion of instructions. Furthermore, this routine helps to prevent the insertion of new errors during the process of correcting old ones.

The 'codecheck' routine examines a program and locates every instruction referring to each location, for example, 537. The references are printed together with their line numbers for easy identification. Thus, one list contains every line of coding which can possibly affect location 537. Codecheck performs this operation for every location from 000 through 999. In general, less time on the computer is required than for a trial run of the program being tested.

B. Hasbrouck
Eckert-Mauchly Division
of Remington Rand

INPUT SCALING AND OUTPUT SCALING FOR A BINARY CALCULATOR

By

E. F. Codd and H. L. Herrick
International Business Machine Corp., New York

Input Scaling

Suppose 1) the input for a problem which is to be solved on a binary calculator is given in decimal form; 2) the programmer desires to specify in his program the scale factors to be applied to intermediate results to keep these results within the capacity of the registers of the calculator; 3) the scale factors to be applied are powers of two (this allows advantage to be taken of the shifting operations which may be built into the calculator).

The following questions may now be asked. 1) Can preliminary scaling of the decimal input (either by hand or on an auxiliary decimal calculator) be avoided? 2) If the binary calculator can be programmed to carry out some equivalent form of scaling, how closely does the converted and scaled input approximate to the