An Optimizing Program for the IBM 650*

BARRY GORDON

Equitable Life Assurance Society, New York, N. Y.

The IBM Type 650 is a stored-program magnetic drum calculator, with input and output on (separate) punch cards. The memory capacity is 1000 or 2000 words, but the program under discussion was written for a 2000-word machine; word length is ten decimal digits plus sign. The 650 uses a one-plus-one-address full-word instruction, consisting of a two-digit Operation Code, a four-digit "Data Address", and a four-digit "Instruction Address". Addresses run from 0000 through 1999 for the 2000-word drum, plus four addresses (8000 through 8003) for the Control Console and Arithmetic Registers. The ability to address these rapid-access "800X" locations permits an increase in calculating speed; however, the primary factor in determining this speed is the latency time arising from the use of the magnetic drum memory.

When locating information on the drum, certain items (e.g. input and output data) must occupy specific locations, due to the nature of the 650. The remaining words—comprising the bulk of the data and instructions—can be located at will, by virtue of the 650's one-plus-one-address instruction. A complex set of rules, relating instruction and data locations and Operation Codes, enables the programmer to choose locations which minimize latency time.

Due, in part, to the complexity of these rules, a minimum-latency program is a very trying thing to write. An abridged set of rules will yield a low-latency program, with somewhat less effort; such a low-latency program will be called "optimum", as the improvement-per-unit-effort almost vanishes beyond this point. However, even optimum programming is a tedious job; in its very simplest form, it requires something like this for each instruction:

Given (N) = the Instruction at Location N:

1. Find the optimum location for the Data Address of (N), based on N and the Operation Code;
2. Find a free location as near as possible (in timing) to the optimum found in Step 1;
3. Place data at location D, found in Step 2;
4. Find the optimum location for the Instruction Address of (N), based on N and/or D and the Operation Code;
5. Find a free location as near as possible (in timing) to the optimum found in Step 4;
6. Place the next instruction at location I, found in Step 5.

This procedure is quite an over-simplification. For one thing, an accurate record must be kept, showing which words (data and instructions) have already been located, so that references to them may be made correctly (skipping Steps 1–3 and/or 4–6). Another record must show, as the work progresses, which drum locations have been used. In addition, those items requiring specific locations must be given those locations before the optimizing procedure is started. Finally,

* Presented at the meeting of the Association, September 14–16, 1955.

the 800X addresses receive special treatment, not included in the skeletal procedure shown. All in all, the writing of an optimum program involves several times the effort of sequential coding, and is subject to many more errors. But, optimizing can triple calculating speeds.

With optimizing being both tedious and valuable, it was only natural to ask "Can the 650 be programmed to optimize its own programs?" The answer to this is: Yes—but certain compromises are necessary. One approach restricts the size of the program which can be optimized, e.g. a limit of 500 words (one-fourth of the machine's memory capacity). Another approach calls for the writing of programs with pseudo-addresses (unintelligible to the 650 itself) by means of which all cross-referencing is done by the programmer. We are thus led to a second question, "How much will we demand of an optimizing program, and where are we willing to compromise?" A general answer is: With a minimum of help from the programmer, an optimizing program should substantially reduce the latency of any program the 650 can execute. Specifically:

1. The latency reduction had to be substantial, but not necessarily the best possible, or even competitive with the work of a programmer;
2. All programs up to 2000 words had to be handled, and in the same way;
3. All cross-referencing had to be done automatically by the machine;
4. Actual 650 instructions had to be input and output, so that programs could be debugged, optimized, and modified any number of times and in any order.

These requirements have been met by the optimizing program now in use at the Equitable Life Assurance Society.

In use, our program makes but one demand: for each D and I, the programmer must say whether it is actually an address or not; then, for each actual address (including N), he must say whether it is an absolute address which already refers to a location, or a symbolic address to which the machine will assign a location. For example, if 70 0001 0642 says "Read a card into Locations 0001–0010 and get next Instruction from Location 0642", D and I are absolute and symbolic, respectively; if 70 0001 0642 is a table value for csc 70° = 1.0642, D and I are not addresses. We show this by means of indices associated with N, D, and I: 0 for no-address, 8 for absolute, and 9 for symbolic. Beyond this, the programmer simply writes a sequential 650 program, after which the machine takes over, paralleling the manual optimizing procedure on each instruction in turn—with less ingenuity than a human, but with greater accuracy. With the optimizing program loaded, the program to be optimized is run through the machine, one word per card; at this time, all absolute addresses are assigned their proper drum locations. On re-running the same program deck, as each symbol is used for the first time, the machine will assign optimum locations and punch a new card for each card read. Note that the new cards will have the same form as the originals, including the address-indices, so that today's output can be tomorrow's input.

To record the use of the machine's 2000 memory locations, and the assignments (for cross-referencing) of the 2000 address-symbols written by the programmer, two tables are used. These occupy 1200 memory locations, as follows:

A Symbol Table occupies locations 0000 through 0999, two entries per location. The table entry in the upper half of location A refers to the symbol A; the table entry in the lower half of location A refers to the symbol $1000+A$. If the entry for the symbol A is 8,0000 (its initial value), the symbol has not yet been assigned a location. A symbol assigned to location Y has a table entry of $9,Y$ where $0000 \leqq Y \leqq 1999$. For example, if (0927) = 8000090519, the symbol 0927 has not yet been used, but the symbol 1927 has been assigned location 0519.

A Location Table occupies locations 1000 through 1199, ten entries per location. The table entry in digit P of location M refers to location

$$500Q + 50(10 - P) + R,$$

where

$$\frac{M - 1000}{50} = Q + \frac{R}{50}.$$

The table entry for location A is digit $10-Q''$ of location

$$1000 + 50Q' + \mathrm{R}'',$$

where $A/500 = Q' + R'/500$ and $R'/50 = Q'' + R''/50$. A table entry of 8 shows a free location; 9 shows a location which is already used (or otherwise unavailable, as the optimized program can be restricted to any portion of the memory). For example, if (1052) = 8888888899, then 0502, 0552, 0602, $\cdots$, 0852 are free locations, but 0902 and 0952 are not. Note that these locations are all equivalent for optimizing purposes.

These tables are the heart of the program, the rest of which fits easily into the remaining 800 words of memory.

In practice, the program has proven extremely simple to use, even where it was necessary to add address-indices to programs written some time ago. Although much of our work is input-output-limited, we have obtained some figures on the program's effectiveness. Originally written sequentially, the optimizing program optimized itself in about $13\frac{1}{2}$ minutes; the optimized version then re-did the job in about $8\frac{1}{2}$ minutes. The punching operation went from 53 to 90 cards per minute; a search of the entire 200-word Location Table (requiring over 4000 operations) went from about 20 to about $8\frac{1}{2}$ seconds; the setting up of the 1000-word Symbol Table went from over 30 to slightly under 8 seconds. Another program optimized by machine was a 50-word sub-routine, which was restricted to 50 consecutive memory locations; in this case, calculating speeds went from 180 to 333 operations per second. In a third application, a lengthy actuarial calculation (involving several table searches) went—on the average—from $14\frac{1}{4}$ to less than 8 seconds. Finally, a sequentially-coded Monte Carlo program went from 163 to 403 operations per second.

Of course, many factors will affect the results in a given case, notably the amount of input and output involved. In certain programs, e.g. some distribution jobs, no amount of instruction-juggling will change the operating speeds; the examples cited above show program operating speeds multiplied by factors of 1.5, 1.8, and 2.5, with sub-routine speeds multiplied by 1.7, 1.9, 2.4, and 3.8. Results such as these—and the ease with which these results were obtained—have led us at the Equitable Life to the belief that manual optimizing is now an obsolete procedure.