Our Experience With The System/360

By Marty Hopkins

FOR OVER eight months, more than forty CUC Programmers and Analysts with varied backgrounds in systems, data processing, and scientific programming have been working with the IBM System/360. Two facts have emerged from the self-teaching experience on this machine. First, no one "knows" how to program the 360 until they code it. Second, it is very dangerous to assume that you know the 360; it is a tricky machine, not because it has a complex instruction set, but in the way functions are linked and dependent on one another.

What follows is derived from our day-in, day-ut experience in coding the 360. If we seem to emphasize the problems, we do this to offer constructive help to those who will be working with the machine.

In a nutshell, CUC found that:

- **THE 360 IS REALLY DIFFERENT AND TRICKY**
- **YOU WON'T BE ABLE TO PROGRAM THE 360 UNTIL YOU CODE IT, BUT**
- **ADVICE WILL HELP, AND HERE ARE SPECIFIC SUGGESTIONS.**

PROGRAMMING STRATEGY

Although the 360 may be less than perfect to program, we have found that it is seldom necessary to have bad or extremely slow code. This often appens on the 7090, when characters are manipu-ated, and on the 1401, when long moves or extensive arithmetic operations are attempted.

The 360 does impose one critical consideration on the programmer—careful planning. There are various approaches to almost every coding problem on the 360; a hasty decision or short-cut trategy might lead to unexpected difficulties. For xample, when a programmer decides to use decimal rather than binary arithmetic to save conversion time, he cannot, conveniently, use the computed quantities for address arithmetic because the internal addressing is binary. The choice of radix (binary or decimal) is even more complex and will be considered in detail after the nature of the registers is explained.

Good programming strategy depends on understanding the registers. There are 16 fixed-point registers, each 32 bits long. The odd and even numbered registers are paired for double-length shifts, multiples, etc. Registers "1" and "2" are implicit operands in some operations, and register "0" is restricted in its use. All the registers can be used as accumulators. You'll find the situation is like the 705-7080, where the ASU's are used as temporaries, counters, etc. A good program must eep many intermediate results in the registers and thus, minimize "saves" and "restores." In addition, you can use the registers as index registers and for linkage.

REGISTER MANIPULATION

A 360 instruction allows only 12 bits for the address. In 360 terminology, those 12 bits are called the displacement. This means only 4096 bytes can be addressed directly. Four bits are provided to specify a base register. The *contents* of this register are added to the 12 bit displacement to obtain an effective address. Some instructions allow the specification of a second register for address modification; this is the index register. Thus, an effective address is formed by *adding* the 12 bit displacement to the contents of the registers specified in the base and index fields. When specified as a base or index, the contents of register zero is zero. So, in order to address anything beyond the first 4096 bytes, a base or index must have been loaded. This is a critical consideration in programming; the more bases loaded, the fewer registers available for indexing or use as temporaries. There is an added difficulty in planning due to the even-odd pair arrangement and the special characteristics of registers "0", "1", and "2".

The register problem also extends to the floating-point registers. The four floating-point registers are separate from the 16 fixed-point registers; therefore, only floating-point operations may be performed in them. This might sound innocent enough until you realize that to float a fixed-point number, you must (1) load it into a fixed-point register, (2) "OR" in a characteristic, (3) store the word in core, (4) load it into a floating-point register, (5) normalize it by adding floating-point zero, and then, finally, (6) store the result. This separation of fixed and floating registers makes it essential that subroutines agree on the registers to be used for passing arguments. In large systems involving many programmers, the choice is not always easy. The wrong decision will cost each programmer many programming steps.

The method you choose for linkage will also affect overall register assignment. The most inefficient code I have seen, occurs in linking to subroutines and the transmission of arguments. Many

techniques are available, but there is no single one which is particularly good in every situation, nor, for that matter, even in a majority of situations. As soon as you have a basic understanding of the), you should consider the various possible methods for linkage and select the best one as each new problem presents itself. I don't think anything else will teach you more about the computer or be as necessary in developing a good programming strategy.

Our programmers have found that register allocation is not difficult if it is planned in advance. It is helpful to break your programs into subroutines, each covered by a base register. Two registers should be permanently reserved for linkage. Data and constants should be kept together for convenient coverage by a base. Large sets of data, such as tables and arrays, which must be addressed via indexing should also be kept separate. You'll ually find it sufficient to reserve one base register for data and one for instructions within any subroutine. Registers should be specified symbolically so when register assignments must be changed, you can do it by reassembly. This is a sound precaution for all programs. We discovered that it is necessary to save one register for an emergency. A small change in logic may cause a complete rewrite in a tight routine because a register is not available. Don't attempt to address first 4096 bytes directly. Some system probably expects to use them, and even if this is not so, the program is useless if it has to be relocated.

DATA LAYOUT

The other broad area where we advise careful ning is data layout. Early in the analysis stage, you must decide if data are to be decimal or binary. You'll notice each radix has advantages and disadvantages. Binary data is more compact, is operated on faster, and can be used for address computation. However, it must be positioned on a word boundry and requires conversion when the input is unedited. Decimal instructions have two addresses; they allow for variable length fields, and they do not require a register as an operand. However, because decimal instructions may not be indexed, an extra base is often needed. Instructions to convert binary to decimal and vice-versa are available. In most cases we have found that binary data produces the most efficient code.

The variable-length "Move" and "Compare" instructions are a great help in coding, and they will operate on binary data. IBM 1400 series programmers should remember that the word length ot contained in the data. When the word size is carried with the data, it must be a binary count or a special control character. The former can be used to modify an instruction; you must scan for the latter. In general, though, you'll find the variable-length properties of this machine are less flexible than that of the 1400 series. This may be a good thing because instruction-controlled word length, as found in the RCA 301 or STRETCH, allows easy debugging and produces good code.

ACCURACY

Those who plan to do heavy computation should be aware of the accuracy problem. A single precision floating-point number has a 24 bit mantissa. The characteristic is a power of 16, not a power of two, so a number is normalized when one of the left hand *four* bits of the mantissa is a one bit. This means up to 3 bits can be lost in normalization; thus, the mantissa may have only 21 bits of accuracy. That is a little more than six digits. The 7090 gives over eight digits. It may turn out that most scientific programs will be done in double precision. They will be almost as fast, even if they use more storage.

Fixed-point accuracy of 31 bits probably is not going to please those who have finally gotten 35 bits from 7090 Fortran IV. If you want more accuracy in fixed-point computation, do not try to use unnormalized double precision unless you have no multiplies or divides, as those operations always normalize. It is possible to get 31 digits of accuracy in decimal operations, but those operations are slow.

In summary, we would say that the 360 deserves careful study. In most cases, the programmer must start from scratch and reorganize his existing procedures to effectively use the 360. The changeover is radical; it is in the order of going from EAM to the 650. Characteristically, the difficulties encountered in overcoming this rather wide gap are laid to "the machine." But this is unfair, for we have found that each time a programmer invents a tricky new technique to bridge this gap, "the machine" looks better and better to him. Ask our people who are already at work on the 360.

MARTY HOPKINS is a Senior Analyst with CUC where, over the past five years, he has worked on several projects in systems programming. These included the design of COBOL compilers for the H-400 and 800, and a FORTRAN compiler for the H-800. Later, he made a feasibility study on translating 705 programs to the H-800. Marty played a key role in the design and implementation of the DCC simulator, a real time computer, and worked on a study contract for a proposed FORTRAN IV compiler. He was in charge of Project A, now completed, on the assignment that gave him his System/360 experience. (*Barbara Lesser* is overall project supervisor, *Betty Burton* was head of Project B, and *Martha Wilensky* heads Project C.) Marty lives with his wife, two children, dog, and cat on a quiet tree-lined street in Bronxville, a suburb of N. Y.