CGTM 50
J. Ehrman
June 1968

Conceivable Extensions to System/360 Assembler Language

There are numerous restrictions and omissions in the current System/360 Assembler Language. Some of these were undoubtedly dictated by the requirement that the Assembler and its Language be the first and most stable of the language processors to be used in the development of software for the many programming systems for System/360. It is the purpose of this note to examine some of these omissions, and to suggest some directions for possible improvements and extensions. As the title implies, my suggestions are tempered by the feeling that it will be very difficult to have IBM generate an extended Assembler Language which does not accept correct programs in the current Language, as defined by the OS/360 Assembler Language specifications, File No. S360-21, Form C28-6514-5.

It is perhaps worthwhile to mention some design criteria that may be helpful to the designer of a low-level processor such as an assembler (or a compiler-compiler). There are many standard operations that are performed by such processors, like character-string matching, table searching, data conversion, and so forth. While all of these features are not simple to open up to the programmer, it should be possible to provide him with easy access to the information that is derived through the use of such facilities. Thus, the Assembly Language programmer may not want to be able to search the symbol table himself, but all the information about a given symbol should be easily accessible through the use of some sort of attribute notation, of which the "L'" notation for the length attribute is a good example. More important, it should be possible for him to have a large measure of control over the environment in which his program is being processed; for example, he might like to change the meaning of various common constructions, and to change them back to the standard whenever desired. A good general rule is that the processor should make no assumptions about what it is the programmer really wants to do, other than to assemble programs with reasonable ease -- the fact that the base language is to be the machine code for System/360 should be the only restriction.

The discussion here will be roughly in three areas: first, a spotty critique of existing facilities; second, some suggestions for elaborations

of existing facilities; and third, some suggestions for new features of the language. Aside from this crude organization, the items will be presented in random sequence, with no order-of-importance implied.

1. One occasionally writes

"NAME CNOP 2,4"

only to find that whether or not any code is generated, the name-field symbol "NAME" is illegal in the instruction. There seems to be no good reason to disallow this construction; since the Assembler always generates NOPR instructions if needed, it would seem consistent to always assign the appropriate value attribute to the symbol "NAME" with a length attribute of 2 and a type attribute of I.

2. The restriction of values of self-defining terms to 24 bits seems arbitrary; but even if there is a good reason in the current implementation, it should be removed in any extensions. For example, it should be possible to write "DC A(X'12345678')" with intelligible results so long as the restrictions implied by the use of a type-A address constant are satisfied. This would mean that one could write something like "DC A(X'03000000'+SYMBOL)" rather than having to write two statements like "DS 0F" followed by "DC X'03',AL3(SYMBOL)". This will introduce no incompatibilities with the linkage editor.

3. It seems unnecessarily restrictive to limit the use of literals to single operands that can appear in DC statements. For example, if the restriction mentioned in (2) above were still present, one might like to be able to write

L 2,=(AL1(X'03'),AL3(SYMBOL))

or

EX 5,=(MVC 0(0,2),0(3))

with the programmer taking responsibility for addressability problems that might arise in cases like the second example.

4. At present, the rules for the use of attributes are quite inconsistent. For example, if one writes "A EQU 5" he then finds that the length attribute of the symbol A is 1, which somehow implies that the length attribute of a self-defining term is 1. If however he has the misfortune to write a term like "L'5" in an expression, the statement is treated

as erroneous. Furthermore, since the length attribute of something is arithmetically equivalent to a decimal self-defining term, there should be no objection to writing a term like "L'L'L'A" where the result would of course be 1. While this may seem silly when viewed this way, one need only consider that it is quite possible to generate such constructions inadvertently in macro expansions; with this extension, writing macros becomes much simpler because the programmer need not test for all the pathological cases that might become sources of error during the assembly phases. Similarly, there is no good reason that the integer and scale attributes of a symbol should not be available during the assembly phases rather than just during the macro phases; one could generate code for scaled fixed-point arithmetic using these attributes in much the same way as one now uses the length attribute in character instructions.

5. The assignment of letters to the type attributes of macro arguments is the source of many restrictions. It is probably correct to state that virtually the only use made of the type attribute is in the testing of arguments of macro-instructions, where one must compare the type attribute to a single character. Since the comparison is done against a constant, there is no reason why it could not have been a decimal self-defining term instead, which would have allowed many further uses for the type attribute, some of which will be mentioned below. As things stand at present, there is only the single letter "L" available for further assignment as a type, whereas there are many other things one might want to find out about an argument. (See item 22.)

6. It is awkward not to be able to define DSECTs within existing DSECTs, so as to have the same relocatibility attributes for more highly structured storage layouts. For example, in a re-enterable program, space may be reserved for a data area and its accompanying DCB, with both being described by DSECTs. It should be possible to include the DCB definition within the main DSECT, so that the same base register can be used to address them both. An example of such an ability appears in the Fortran H "STRUCTURE" statement, where several different storage layouts can be superimposed.

7. There doesn't seem to be any compelling reason to treat statements such as "L 8,2(0,3)" as implying alignment errors; if the programmer

has said nothing about the uses of the registers, then the Assembler should make no assumptions about what will be in them.

8. The rules for the resolution of expressions into base-displacement form are very carefully laid out in the Assembler Language manual, so that it is clear in writing something like "L 9,8(3)" that the Assembler must resolve the expression (in normal circumstances) with a base of 0, a displacement of 8, and an index of 9. However, a casual perusal of almost any microfiche listing of IBM software reveals that this form of statement is commonly used where the index digit was intended as the base digit. Now the addressing arithmetic of System/360 is simple enough that the generated address is the same in both cases, but the machine user is required to pay the penalty of the additional indexing cycle every time the instruction is executed. In such cases, if (1) there is no USING statement if effect that specifies an absolute expression, and (2) the base digit of the addressing expression is zero, and (3) an implied address is used, then the index digit should replace the base digit. This should yield improved performance of all software on the smaller models of the 360 line, until such time as coders can be retrained to write the operands correctly.

9. The count attribute of a macro-instruction operand should be useable anywhere that a self-defining term may appear, rather than merely in SETA expressions or relations.

10. Substring notation can presently be used only in SETC statements; this often imposes a severe handicap on the reduction of macro arguments which are longer than 8 characters, since the argument must be picked apart in pieces convenient to the macro processor rather than those convenient to the macro programmer. (See the SAVE macro definition for an example.)

11. Since the length attribute of an expression is defined at assembly time to be that of the leftmost term, the same should apply at macro time, so that a macro argument "3+A" will be as acceptable as "A+3".

12. It is a most unusual feature of the F Assembler that an "END" card is logically equivalent to an "end-of-data-set" on SYSIN; I know of no other processor within the Operating System that makes this restriction. There should be the equivalent of the PL/1 "Process" statement or something similar to allow more than one assembly per job step.

13. It would be useful to be able to finagle the input parameters of a macro-instruction as though they were locally defined character variables. That is, one should be able to write

```
        MACRO
        ANY     &X
        ...
&X      SETC  something
```

without having to break the input string down into little pieces contained in local variables. This implies that the symbolic parameters are simply initialized to the values given them by the programmer who uses the macro-instruction.

14. There should be some scheme for qualifying symbols, so as to allow the programmer to specify a structure for his program more detailed than that imposed by the System/360 CSECT structure. For example, one possibility would be to allow qualification similar to that used for PL/1 structures: if one specified that all symbols following were to be qualified by the characters "BLK", then they would appear in the form "BLK.SYMBOL" where it is understood that the symbol may already be subject to lower (higher?) levels of qualification. This also implies that the current restriction of identifiers and symbols to 8 characters should be removed for anything that is not externally defined.

15. There seems to be no compelling reason not to have multiple location counters for each control section, as well as the ability to define CSECT origins arbitrarily.

16. It is a great weakness of the current form of the Language that each specification of a register in a USING statement overrides the previous definition of the value to be assumed for resolution purposes. There should be some means to save and restore the contents of the USING table both for individual registers and for the table as a whole, for an arbitrary number of levels of saving.

17. The same feature should apply to the saving and restoring of the conditions implied by the PRINT statement: it should be possible to specify new parameters without losing the old ones.

18. There should be some means of specifying that the Assembler is to list the location and first line of generated code in macro expansions.

19. What's wrong with unary operators? One should be able to write "A-2" or "-2+A" with the same results.

20. A major omission at present is any facility for specifying remote assembly of code or other statements. The "RMT" pseudo-op of past assemblers was a crude method for performing such remote assembly; it should be possible to specify several ways in which the text being collected for later processing is to be retrieved. That is, one should be able to specify that blocks of statements are to be placed either in named queues or stacks (FIFO or LIFO lists) for later retrieval. The use of queues (by something like RMTQUE/RMTEND) is the more familiar form, but the ability to stack statements (via something like RMTSTK/RMTEND) would allow the coding of things like looping instructions with a minimum of additional effort. By naming the lists on which the text is being saved, the programmer can selectively retrieve any or all of the data saved.

21. The conversion of the values of SETA symbols to characters via a SETC instruction is not very general; there are times when one wants the value in some other representation, such as base 16. There should be some scheme for converting from one representation to another. For example: the statement "&A SETA 19" followed by the statement "&C SETC '&A' " gives &C the value '19', whereas we might have wanted the hexadecimal representation of the value of &A. Thus what is needed is something of the form "&X SETX (&A,16)" which would give a value '13' to &X. Furthermore, such a facility could be used to convert the character values of the type attribute to numeric quantities (for example, "&A SETX ('8',10)" could give &A the value 248) for more meaningful types of tests, assuming that the type attribute can be elaborated as described below. There are some macros in MACLIB in which herculean effort is expended to do the conversion from decimal to hexadecimal, so this suggestion is nontrivial to IBM.

22. The type attribute is currently limited to providing a very narrow description of the appropriate argument: one can determine that something is a self-defining term, or a symbol which names one of a class of statements, or is null, or is otherwise undecipherable. There should be a much larger scope for the type attribute (or perhaps a new attribute called the class attribute) which would allow the programmer to determine for example that a given parameter or character string is a legal symbol (in which case he could further inquire as to what sort of statement it names, as with the current type attribute), whether it is a valid expression, whether it is

a valid operation code, whether it is a currently defined macro name, and so forth. While these are rather broadly-defined constructions, there is also a need at a very low level for a means to determine that a given character is a letter, without having to filter the character through 26 AIF tests as at present. The presence of a conversion feature like that described in item 21 would alleviate some of the problem, since if for example letters were assigned equivalent values 10 to 35, one could test for a letter with "&X SETX (&CHAR,EBCDIC)" followed by "AIF ((&X GE 10) AND (&X LE 35)).LETTER". And one should be able to determine if a character is one of a given set of characters, as for instance with something like "AIF ('&CHAR' IN 'AEIOUY').VOWEL". The current limitations of the character-manipulation facilities of the macro passes should be removed so as to include many simple functions used in character-string processing.

23. The current restrictions on the macro-time uses of attributes to quantities in the argument list of a macro were undoubtedly derived from the necessity of limiting the size of the local dictionaries for each macro. There should be no need for this if the programmer is willing to pay the price of having the full symbol table available when he wants it; this would then permit such useful activities as determining the properties of generated symbols, which is currently impossible.

24. To extend the above argument, it should be possible to determine the attributes of a symbol or other quantity as soon as it is defined. For example, the statement "X EQU 5" is a complete definition of the symbol X, so that its attributes should be available at any time following that statement. For symbols whose attributes have been multiply defined, there should be a means to determine that this has occurred (by using, for example, the boolean "M'" attribute notation), but otherwise the assignment of a value to the other attribute requests should be just the current value of the attribute in question.

25. It should be possible to use a macro to generate other macro definitions; it should be possible to define a macro anywhere before it is used; it should be possible to redefine macro definitions; and it should be possible to save and restore definitions, about which more will be said below in item 30.

26. There should be some means to control the collection of symbol table and cross-reference information. For example, if there are many DSECTS at the beginning of a program, but only a few of the symbols defined thereby are actually used in the following text, it would be helpful to be able to turn off the cross-reference collection for the duration of the definitions, and then turn it back on after the definitions are complete. It is assumed that any symbol-table entry would retain the number of the statement in which it was defined.

27. The types of list structure available for macro operands should be expanded to include arbitrary lengths and nestings.

28. There appears to be no good reason why keyword parameters in a macro-instruction statement should not be allowed to appear anywhere, including the name field.

29. It would be useful in many macros with arguments that can take on a large number of different character values to be able to generate sequence symbols: for example, the statement "AGO .A&B" could be used to speed up the branching process through a complex filter.

30. This is probably the most involved of the suggestions. One of the major limitations of most assembly systems is that the programmer is presented with a fixed set of operation codes and a fixed set of rules for converting them into relocatable text, and is expected to live happily within those boundaries. As an example, suppose that one would like to be able to write a set of macros which have the same mnemonics as the instructions that affect the general registers, and keep track (through the use of some global variable symbols) of the integrity of registers previously defined as base registers through USING statements. Such a scheme would be helpful for beginning students: the macros could perform a crude check for the safety of the base registers at assembly time, and then go ahead and generate the desired instruction if no obvious errors are found. This sort of procedure will require that one be able to redefine the machine instruction codes as macros for parts of the program, but be able to retain the usual processing also. There should therefore be a general facility for stacking various quantities of interest to the programmer, through the use of a PUSH and POP facility. This is implicit in items 16, 17, and 26 above, but the extent of the application there is very narrow. It should be possible for example to PUSH the definitions of any or all of the following:

(1) individual operation codes, (2) classes or groups of operation codes, (3) segments of the symbol table (which would allow the programming of block-structured material with great simplicity), (4) programmer macro definitions, (5) system macro definitions. An example of the latter might arise in a case where a programmer has written a segment of code assuming the usual definition of the SAVE macro, which is to be followed by a segment in which he uses his own definition. He should then be able to restore the previous definition by writing something like "POP OPCODE=SAVE". It seems that the following are reasonable rules for the use of the PUSH and POP facility:

1. When an operation code has been pushed, it is undefined. Clearly there should be a facility like OPSYN so that PUSH and POP can be themselves pushed and popped, and so that parallel uses of instruction codes are possible.
2. If a symbol has been pushed, it is undefined until redefined. If it is then popped, it reverts to its previous definition status.
3. The basic operation codes are "top-level" in the sense that they cannot be popped past the top level. An attempt to do so will generate a diagnostic, but will not affect their status.

Some other applications of such a feature might be as follows:

1. The owner of a machine without a decimal feature or a floating-point feature could generate code that would respect this fact by executing the statement "PUSH OPCODE=DECIMAL" which would cause all the decimal instructions to become undefined. He could then replace them at his discretion with macro definitions.
2. The owner of a model 85 could make the extended floating-point mnemonics available by coding a set of macros; but it would be much faster if they were included in the assembler to begin with, and could be used by initially specifying "POP OPCODE=EXTENDEDFLOAT", with the default option being that the opcodes are undefined initially.

31. One final item which involves the linkage editor: at present it is possible to structure only a single virtual (external) dummy control section through the use of the DXD instruction and the Q-type address constant.

It would be most helpful if it were possible to define for link-editor resolution several different and distinct virtual dummy areas, in much the same way as the DSECT instruction can be used to define many virtual areas whose values are resolved (for base-displacement purposes) at assembly time. For example, one might want to specify that the symbols A and B are to be resolved by the link editor in different virtual areas (numbered 1 and 2) by using something like "DC Q1(A),Q2(B)", and the lengths of the resolved virtual areas could be specified with something like "CXD 1" and "CXD 2", with the accumulated lengths for each being placed in the correct location. This of course implies the generation of something like a "hyper-relocatability attribute", but it introduces no new ideas into either the assembly or link-editing process.

32. It should also be possible to specify addends for the operands of Q-type address constants. At present, one must either name all the items in the virtual area unless they are in a DSECT; it should be possible to prescribe one's own levels of structure by being able to write something like Q(A+4) with sensible results. The Linkage Editor should therefore do the same as it does for A-type adcons, and simply add the value it calculates for A to the value 4 provided by the text for the Q-type adcon.

33. It should be possible to determine the length of a CSECT or DSECT. This would best be done at Assembly time by making it the length attribute of the symbol naming the section, and at link-edit time by something like the CXD instruction, where the Linkage Editor fills in some known piece of information.

34. There should be System Variable Symbols &SYSDATE and &SYSTIME to return the current date and time respectively.

35. There should be something like a computed GO TO in the macro passes, so that one could write "AGO (.A,.B,.C,.D)&SWITCH".

36. It seems reasonable to allow logical operators in arithmetic expressions.

I wish to thank the many who provided suggestions for this note.