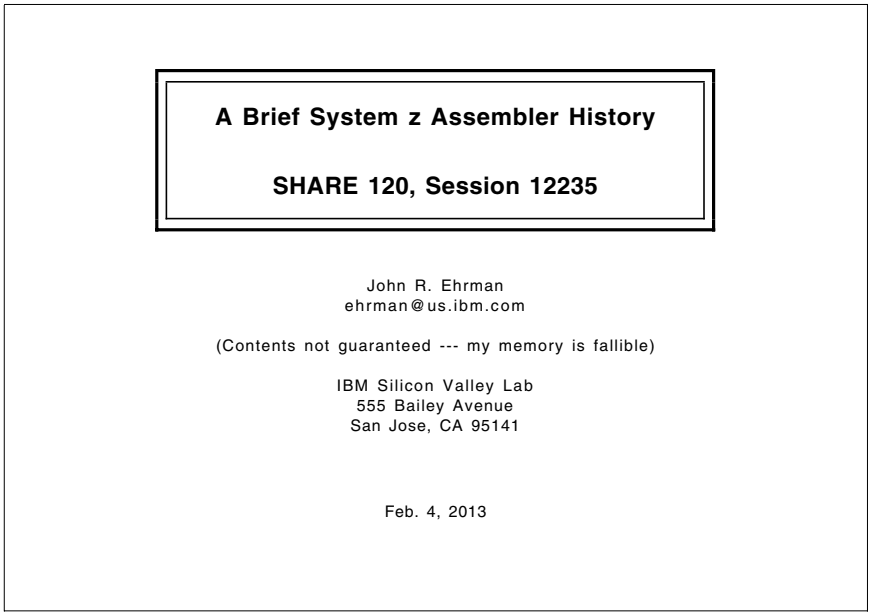
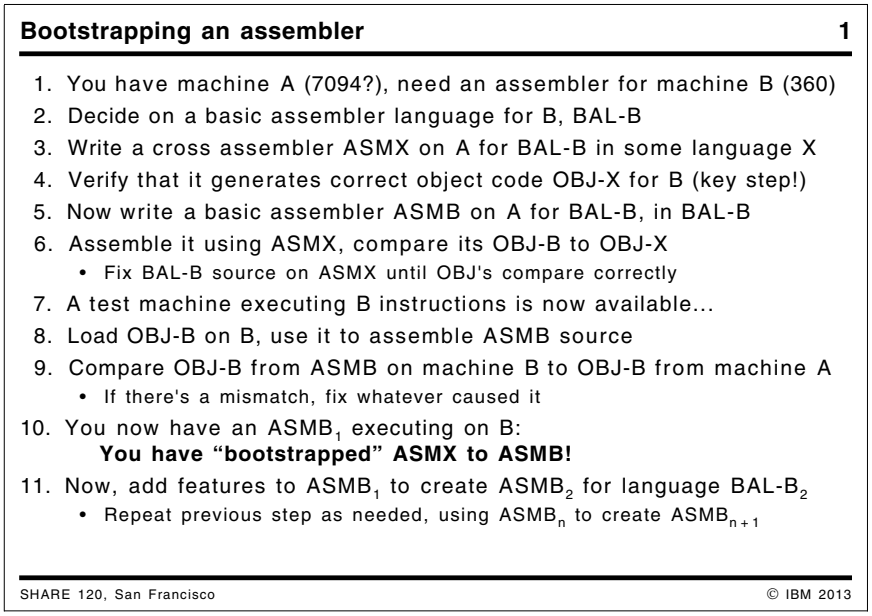


A Brief System z Assembler History

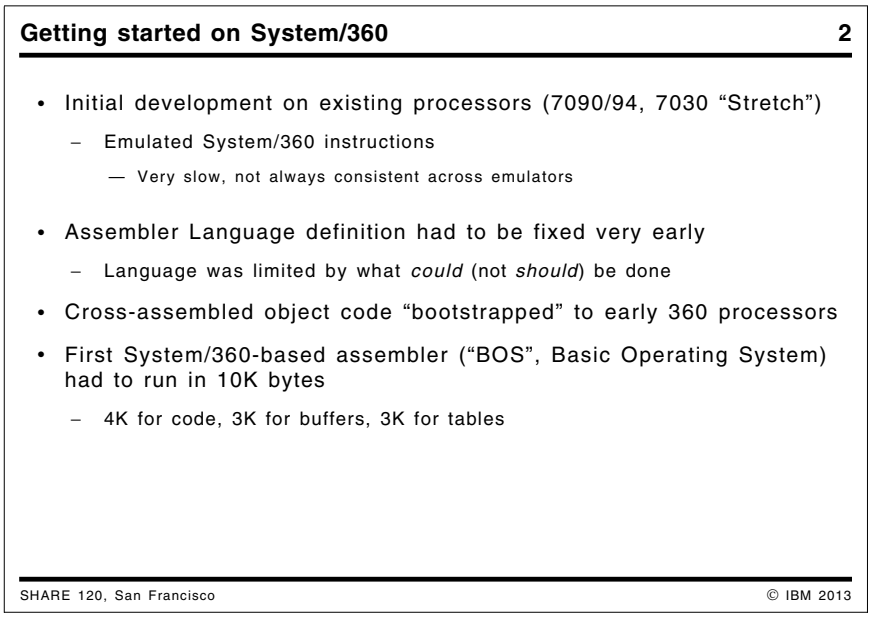
The System/360/370/390 Assembler Language[1] was created almost in parallel with the design of the original System/360 architecture, because the assembler was expected to be almost the only program development tool in the initial stages of software development. Existing machines such as the 7094 emulated the new architecture, and a "cross-assembler" generating the new machine language was used to create test cases, basic tools, and the initial stages of the software "bootstrapping" process.

[1] Some of this material is drawn from:

- The proceedings of the *IBM Macro Assembler Conference*, May 1-3 1967, Los Gatos, CA
- The SHARE presentation *"Assembler Design and its Effect on Language and Performance"* by H. Joseph Myers of IBM San Jose, at SHARE XXXIV in Denver Colorado, March 1970
- *IEEE Annals of the History of Computing*, Vol. 19 No. 4, pp. 47-49



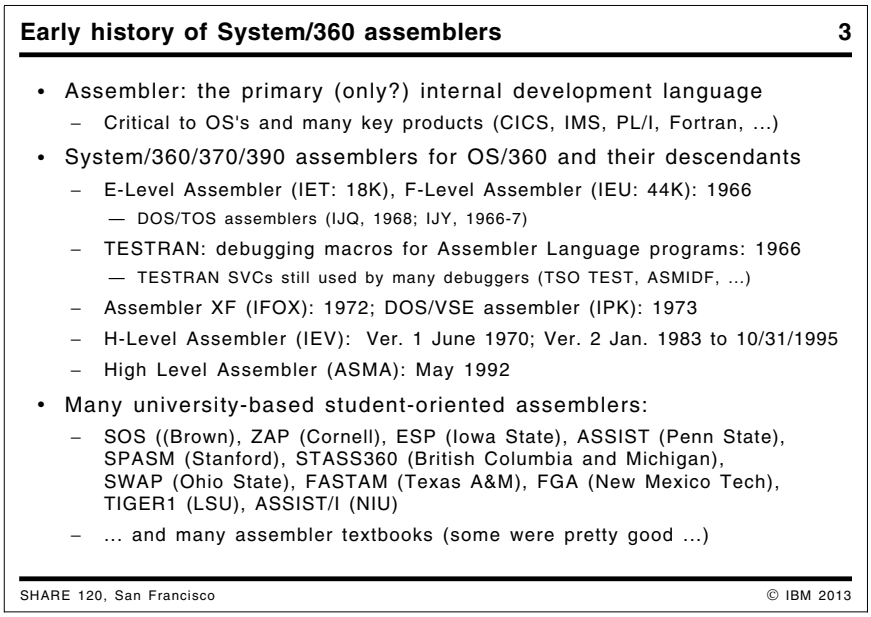
Bootstrapping was widely used at the time System/360 was being designed, because the rapid evolution of the computer industry meant that dozens of different machines and architectures were created. Generating software for a new processor usually meant using an existing processor to generate the initial and basic programs to run on the new machine.



The System/360/370/390 Assembler Language borrowed useful ideas from contemporary assemblers (such as IBMAP for the 7090) and added several major innovations. Among these was the concept of a conditional assembly language separate and distinct from the "ordinary" assembly language from which machine-language object code is generated. The concepts of conditional assembly and macro instructions were widely used at that time, but typically the conditional language facilities were intermixed with the ordinary language: for example, the values assigned to ordinary symbols might change many times during an assembly; there were no distinct variable ("SET") symbols.

The decision to support different classes of symbols — ordinary symbols for the ordinary assembly language, and variable and sequence symbols for the conditional assembly language — permitted greater conceptual clarity in both languages as well as greater functional richness in each.

The pairing of the languages is unusual: the separate and distinct conditional and ordinary languages are integrated in ways that give you much greater power and flexibility in writing programs. No other programming language supports its incredibly powerful coupling of the outer (conditional) and (inner) ordinary languages.



Early History of System/360 Assemblers

The memory sizes of System/360 machines was specified with letters: E meant a 32K-byte machine (14K bytes for the system, 16K bytes for applications); F meant 64K bytes (20K for the system), G 128K, and H 256K (56K for the system; 256K was a very big machine in the 1960s!).

Assembler Language was the first programming language available for System/360, so it was critical to the entire software development effort.[2]

Early assemblers were constrained by limited memory sizes; because there were no interactive[3] debuggers, the TESTRAN system of tracing and debugging macros was developed for Assembler Language programs. Many of its basic capabilities are still used in today's debuggers.

The rapid acceptance of System/360 machines encouraged universities to teach is programming languages (“academic discounts” also helped). Because turnaround on these systems tended to be slow (student pro-

[2] Among the important uses of the Assembler was system generation, or “SYSGEN” for short. This was done in two stages: in Stage I the configuration of the machine and its software was described in macro calls which when assembled generated a large job stream. Stage II ran those jobs to build the system; the full SYSGEN process often took a full weekend.

[3] The Allen-Babcock company created an interactive PL/I like interactive system using modified microcode to implement specialized instructions.

grams are rarely error-free), many universities created special student-oriented fast translators for Assembler Language, Fortran, PL/I, and other languages.

Assembler Processing

We will first describe some aspects of the implementations of ASMF and ASMH, their effects on the language, and then some further extensions made in High Level Assembler. Some definitions:

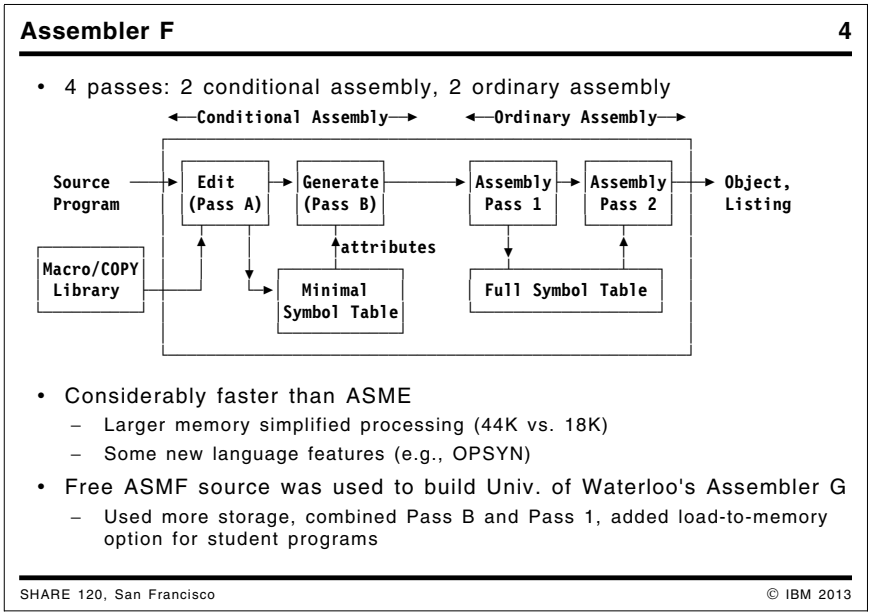
- We say that an assembler makes a “pass” over the source program when the program to be assembled is read completely. A fundamental difference in the designs of ASMF and ASMH is the number of passes each takes to complete an assembly.
  - A “phase” is a subset of a pass, where different sets of instructions are loaded to process portions of the data created by previous phases. For example, an initial phase of Pass 1 might read the source program, identify the fields of each statement, compress them, and then write the results to auxiliary storage.
- An “interlude” is a pass over some internal tables, and usually takes place between or after source “passes.” An example of interlude processing is the sorting and printing of cross references. (We also talk about a “postlude,” which is an interlude after the last pass.)
- By “binding” we mean the substitution of a value for a symbol. Substitution of X'D2' for the MVC mnemonic is one example of binding. The decision that a macro dictionary will be a particular size is also a binding process. Early binding, in general, improves efficiency and decreases flexibility.

Traditionally, the assembly process is described in two passes:

Pass 1: Determine the amount of object-code storage needed by each statement, adjust a “Location Counter” accordingly, and assign location values to symbols; note all external symbols.

Pass 2: Use the information collected in Pass 1 to generate object code.

Because the base of the overall design of today's High Level Assembler (“HLASM”) is based on Assembler H (the “H-level assembler,” or “ASMH”), some of the power and flexibility of the language supported by High Level Assembler comes from implementation choices made in the design of ASMH. ASMH's predecessors, the E-level assembler (“ASME”) and F-level assembler (“ASMF”) were constrained by the small size of early System/360 processors, and many features (and limitations) of today's Assembler Language can be traced to the original design of ASME and ASMF. (ASMF was a repackaging of ASME to use more storage and fewer phases, but the logical design was the same.)



Assembler F

Figure 1 shows a simplified picture of the 4-pass structure of ASMF. (Actually, the four "logical" passes of ASMF required seven physical phases; ASME required 23 physical phases.)

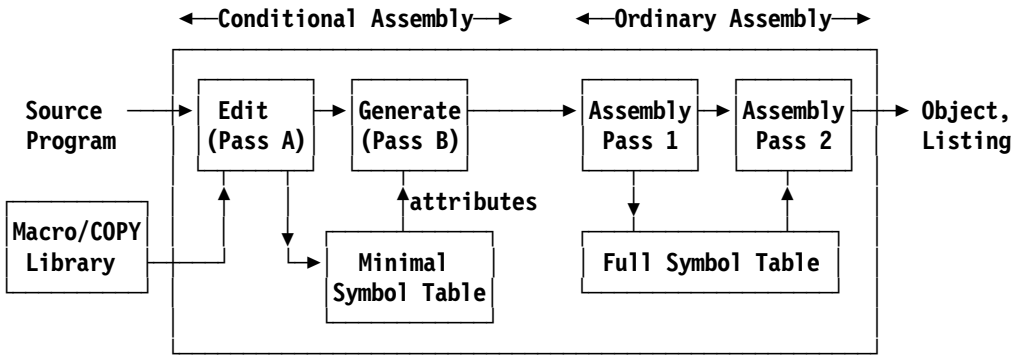


Figure 1. Assembler F Processing

In Figure 1, the first ("Pass A") is called "Edit": this converts the source program into an internal form more convenient for processing. A special kind of editing, "macro editing" converts macros into an internal form and also constructs macro dictionary descriptors that define the assembly-time storage layout for variable symbols. The fixed-size local dictionary descriptors for macros and for open code, and a global dictionary for global variable symbols, are constructed during this "editing" process.

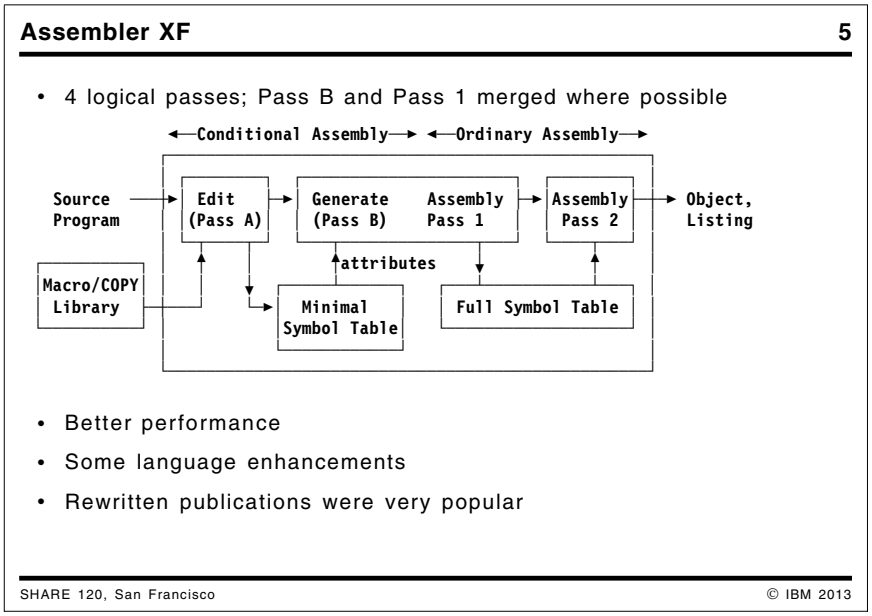
In addition to the dictionaries, the assembler must build a "symbol table" of ordinary symbols in which to record symbol attributes. In ASMF, two symbol tables are constructed: one during conditional assembly for macro processing (to collect symbol attributes of macro arguments, as needed for macro generation), and one for ordinary assembly (for all symbols). (ASMH uses a single symbol table for both purposes; we will discuss the implications of these designs below.)

The term "Generate" is used to describe the ("Pass B") macro and conditional assembly activity that substitutes macro-generated statements for the occurrence of a macro name in the source program. This includes

the substitution of values of variable symbols where they appear, and also applies to the interpretation of the AIF and AGO operations controlling conditional assembly. During this "generate" pass of ASMF, all conditional assembly statements are executed.

The output from the "generate" pass is the completed source text to be assembled by the usual two passes of ordinary assembly. The only remaining vestiges of conditional assembly are in the listing and diagnostic records that will be written during the final assembly phase ("Pass 2").

This is not to say that the two assembly passes are simple! If storage is limited, a large program with many symbols may require multiple iterations over the edited source to collect and then substitute symbol values, discarding the symbols no longer needed to make room for the next set to be collected.



Assembler XF

Figure 2 illustrates the structure of Assembler XF (also known as the "IFOX" or "IFOX00" assembler). Its main feature was to combine the second pass of conditional assembly ("Pass B") and the first pass of ordinary assembly ("Pass 1").

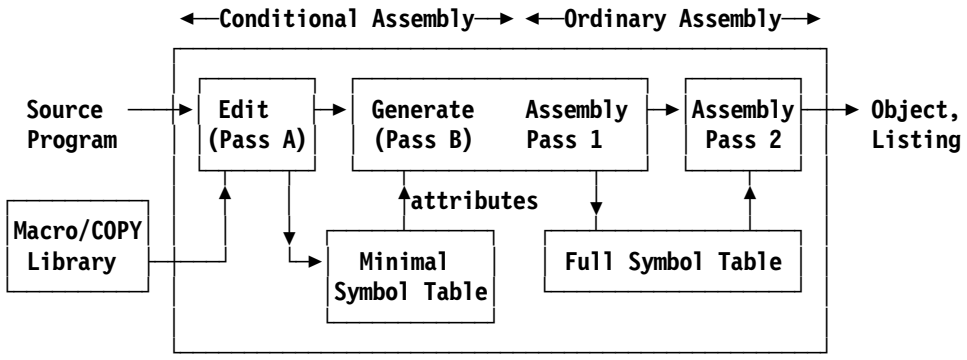
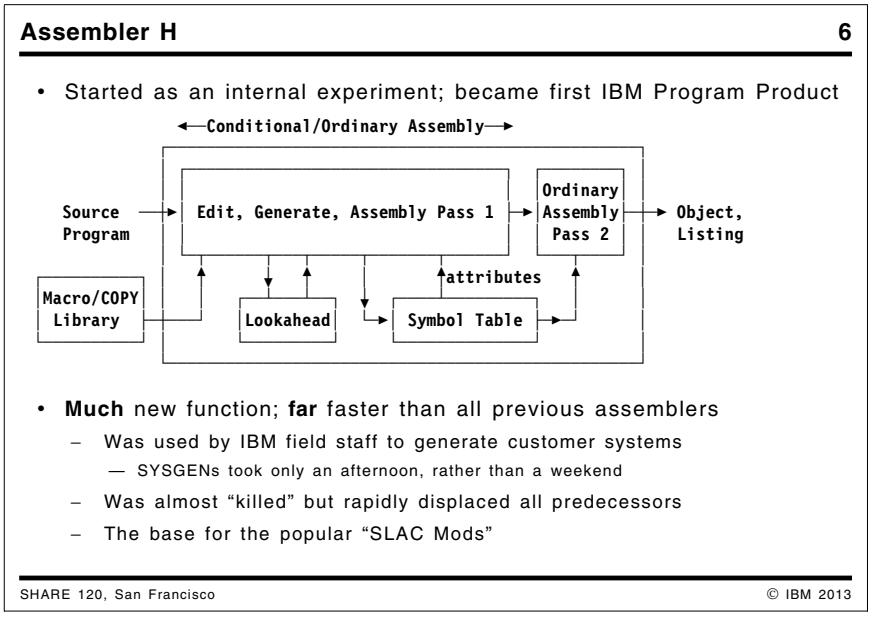


Figure 2. Assembler XF Processing

The primary benefits of Assembler XF were improved performance, and some language enhancements made at the same time. (And, the assembler manuals were completely and elegantly rewritten!)



Assembler H

Figure 3 illustrates the structure of ASMH; this is also the structure on which HLASM is based, so much of this description of ASMH also applies to HLASM.

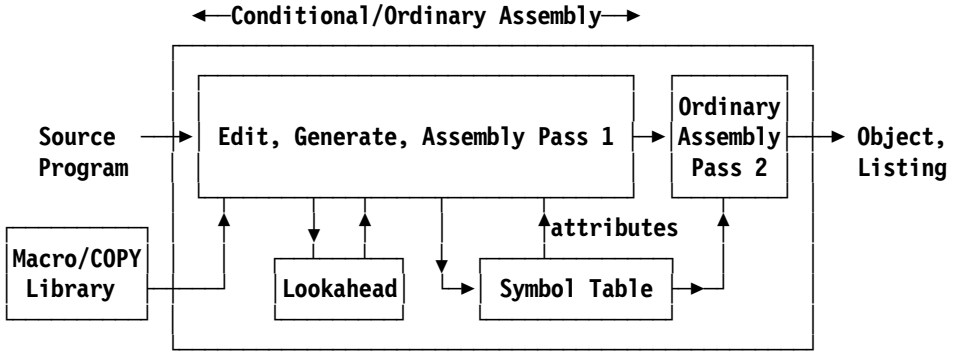


Figure 3. Assembler H Processing

ASMH combined the first three passes of ASMF into a single pass, with a special "lookahead" feature required to support some of the language capabilities of ASMF.

- ASMF is a four-pass assembler, while ASMH and HLASM are two-pass assemblers.
- ASMH and HLASM perform in the first pass the functions performed in the first three passes of ASMF.
- ASMF binds the size of macro dictionaries during the edit pass, while ASMH and HLASM never bind them.

Because of its small storage limit, ASMF was organized so that much of its working data was held on external storage media in three work files. All edited macros and their local dictionaries are maintained

externally, as is the edited source program that is sent through the various passes. The global variable symbol dictionary and portions of the symbol table are kept in storage.

ASMH, on the other hand, managed all of its working storage with its own paging system and a single external data set to handle spillage. Because ASMH was designed for larger (but still constrained) real storage environments, it writes ("spills") its working-storage "pages" to the external utility data set. HLASM can utilize today's much larger virtual storage environments, and does not write its "pages" to the utility file unless more central storage is required than was provided; HLASM's ability to reduce I/O activity helps to improve overall performance.

Some language enhancements are due either to the availability of increased storage, or from experience with the limitations of ASMF. For example:

- Symbols up to 63 characters long (vs. 8 in ASMF)
- Macro definitions can appear anywhere in the source before they are called (vs. at start of source in ASMF)
- Macro definitions can be redefined (not allowed in ASMF)
- Macro definitions can be nested (not allowed in ASMF)
- Macro calls can mix positional and keyword arguments (not allowed in ASMF)
- SETC variables up to 255 characters in length (vs. 8 in ASMF)
- Computed AGO and AIF (not supported in ASMF)
- Multiple values assigned in SETx statements (not supported in ASMF)
- Unbounded variable symbol arrays (fixed-size arrays in ASMF)
- Created variable symbols (an extraordinarily powerful concept, allowing "associative" naming, access, and search)
- Nested sublists (not supported in ASMF)
- SETC duplication factor (not supported in ASMF)
- Attribute references to arbitrary variable symbols (references in ASMF limited to parameters and ordinary symbols)
- PUSH USING,PRINT (not allowed in ASMF)
- OPSYN for mnemonic redefinition (not allowed in ASMF)
- Term values extended to 4 bytes (vs. 3 for ASMF)
- Labeled COMmon (vs. only unlabeled in ASMF)
- The LOCTR instruction for better program organization (not allowed in ASMF)
- Enhanced diagnostics, shown inline (vs. at end of listing for ASMF)

At the time ASMH was developed internally, it was anticipated that PL/I would supplant all other programming languages (especially Assembler Language!) ASMH was very popular among selected customers and also inside IBM (see the "SYSGEN" footnote on page 3) which helped ensure its survival. Its power as a macro-based cross-assembler was demonstrated when it was used to generate object code for the forthcoming 1800 machine before the hardware was built!

We will now describe processing differences between ASMF and ASMH.

Edit Activity

In the edit pass of ASMF, macro dictionaries are prepared by creating fixed descriptors and dictionary space for each variable symbol, as declared on LCLx and GBLx statements. ASMH handles all declarations dynamically, as each macro is expanded during a generate operation; this allows extension of the language in several directions.

First, variable-symbol arrays need not be a fixed size as they were with ASMF. The blocks of array elements are chain-linked together as they are required; such an arrangement could not be tolerated in the small-storage design of ASMF.

Similarly, the space allocated for SETC variable symbols can be varied according to the actual dynamic requirements, unlike the fixed length of 8 characters used for SETC variables in ASMF. ASMH supports SETC variable symbols up to 255 characters in length (1024 for HLASM) and dynamically allocates storage to them as it is needed. (Garbage collection — recovery of unused space — for local variable symbols occurs automatically upon exit from a macro.)

A second extremely powerful language feature is the availability of "created variable symbols" that let you create associative symbol tables during conditional assembly.

The editing of library (or system) macros is handled differently by ASMF and ASMH. In ASMF, all operation codes which appear in the source program and which have not been recognized already are placed on a list. When the END statement is reached, the names on this list are used to search the macro library. As each name is found, the corresponding macro is edited and its name is removed from the list. The editing of library macros may add new names (such as inner macro calls on other library macros) to the list. Processing continues until each name has resulted in an edited macro definition or has been classified as an undefined or illegal operation code. With this approach, all macros are edited that *could eventually* be called. It is possible as a result of AGO/AIF branching during the generate pass that all calls on a particular macro may be bypassed. ASMF may therefore edit macros from the library that are never used!

ASMH, on the other hand, does not search the macro library until it is actually presented with a call on that macro. Upon finding an unknown operation code, generation is suspended until the required macro can be retrieved from the library and edited; this is possible because of the co-residence in a single pass of the edit and generate routines. This method of handling library macros is called "demand editing."

During the edit pass of ASMF, attributes of *visible* symbols are collected and placed in a symbol table. (Thus, the attributes of *generated* symbols can't be known to other macros.) At the same time, basic assembler statements are partially edited, while conditional assembly statements are completely edited. Furthermore, ASMF retains as part of the statement record the source record image, which is passed through the assembler three additional times so it can be used in the final assembly listing.

In ASMH, attributes are collected and all statements are completely edited. Macro calls, conditional assembly statements, and statements needing variable symbol substitution are passed directly to generate processing. All other statements are sent directly from editing to Pass-1 assembly processing and no original source statements are kept; the assembly listing is created by reconstructing the edited statements.

Generate Activity

The input to the generate pass of ASMF ("Pass B") is partially-edited source text. The pass also has the symbol table with attributes and the global variable symbol dictionary available in storage, and various macro dictionary descriptors and edited text on external files. At the end of the edit pass ("Pass A"), an interlude operation has compressed the symbol table so that *only those symbols which appeared as macro arguments remain*. This space-saving operation unfortunately prevents symbol attributes from being passed to inner macros unless they are explicitly passed from an outer macro. A second consequence of this "compression" interlude was that all macro names were discarded, preventing subsequent substitution of macro names for operation codes.

The generate pass "executes" (that is, interprets) conditional assembly statements (AGO, AIF, SETA, etc.) and sends generated statements to the next pass.

In ASMF, whenever a macro call occurs, the corresponding definition is brought from the external storage where it was placed by the edit pass, and expanded. In ASMH, the larger storage design allows many of the macro definitions and their dictionaries to be kept in central storage.

Lookahead Mode

The System/360 Assembler Language presents a difficult problem when implementing a two-pass assembler like ASMH: the problem of forward referencing of attributes during conditional assembly. ASMF collects these attributes during the edit pass; ASMH solves this problem in a manner analogous to the demand editing of macros. That is, an extra step is taken only when a forward attribute reference is actually encountered. This is called “lookahead,” and is not necessarily a full pass.

Whenever an attribute of an unknown symbol is required during a conditional assembly operation, ASMH suspends operation and enters “lookahead mode.” The input stream is scanned until the required symbol is found. Its attributes are then entered into the symbol table and normal processing is resumed from the point where it was suspended. Whenever lookahead mode is invoked, the source text is compressed and saved. Also, attributes of all symbols encountered over during the search are entered into the symbol table, so that those symbols become “known” and won't cause lookahead. Naturally, further requests for statements from the input stream will be taken from the lookahead file until it is exhausted, when input then resumes from the primary (SYSIN) file again.

In addition, lookahead mode can switch to COPY code from the library, and to edited macro text, allowing for implementation of nested COPYs and nested macros in any mix.

Some language enhancements also profit from this combination of edit and generate activity. In ASMF, all macros must be defined before any can be expanded. In ASMH, macros can be redefined, or even defined within macros. Because symbols produced by generation are dynamically added to the symbol table, they are available for use in subsequent conditional assembly. (Some symbols may have been entered into the symbol table during lookahead; their attributes will be updated to their “correct” values, if necessary, when they are generated.)

ASMH Assembly Pass 1

ASMF performs assembly Pass 1 in the traditional way. Partial editing done during the edit pass has isolated the name, operation, operand, and comment fields, and the operation code has been bound. In Pass 1 a separate location counter is maintained for each control section, and symbols are collected from the edited text and placed in the symbol table. Note that this is the second time that some of the symbols are collected; this is because some symbols were missing from the edit-pass symbol table (because they were not involved in macro calls, or were generated from conditional assembly statements or macros). ASMH avoids this problem by keeping a single symbol table, not compressing it, and by not binding its size.

Unlike ASMF, ASMH allows forward referencing of symbols by EQU operands and operands of other types of statements that can affect or depend on the location counter (for example, ORG and CNOP, and the length and duplication modifiers of DC and DS statements). Figure 4 illustrates this.

```
        ORG   *+A       ◄── forward reference to A
        CNOP  C—2,8     ◄── forward reference to C
        BALR  14,15
A       EQU   L'B       ◄── forward reference to B
B       DC    XL(C)'1'  ◄── forward reference to C
C       EQU   4
```

Figure 4. Example of Forward Referencing of Symbol Values

These language features are implemented by the use of a “pseudo-op file” that keeps track of location counter discontinuities. Each time a discontinuity occurs (during Pass 1) due to an unknown symbol or alignment constraint, a new entry is made in the pseudo-op file that represents a temporary internal location counter for the next segment of code. Thus, the location counter in Pass 1 can actually be a series of small

disjoint location counters that must be combined when the discontinuities are resolved in the interlude at the end of Pass 1. This lets you define your own location counters with the LOCTR instruction.

During Pass 1 of ordinary assembly, ASMF makes a further edit of the operand fields. ASMH does all editing once, and retains all operation codes in a loosely bound state. This allows macro and operation code redefinition to occur easily. Figure 5 shows a macro definition of READ which has the effect of redefining the system macro READ so that its expansion is surrounded by some auxiliary code, here represented by two MNOTE statements. The first use of OPSYN defines READX to be the same as READ, and the second causes READ to become undefined. (Note that this "undefinition" of READ will not take place until the expansion of READ has started; it will not adversely affect the expansion text.) As the expansion continues, the first MNOTE is processed and then the call on READ is encountered. At that point READ is not defined, so the macro library is searched. The system READ macro is edited and expanded, followed by processing of the second MNOTE. Finally, the last OPSYN restores the definition of READ to its original value.

```
         MACRO
         READ  &A,&B,&C
READX    OPSYN READ       Define READX to be same as READ
READ     OPSYN ,          Undefine READ
.*
         MNOTE 'First inserted statement (pre-process READ)'
.*
         READ  &X,&Y,&Z   Will be brought from Library
.*
         MNOTE 'Second inserted statement (post-process READ)'
.*
READ     OPSYN READX      Redefine READ to this macro
         MEND
```

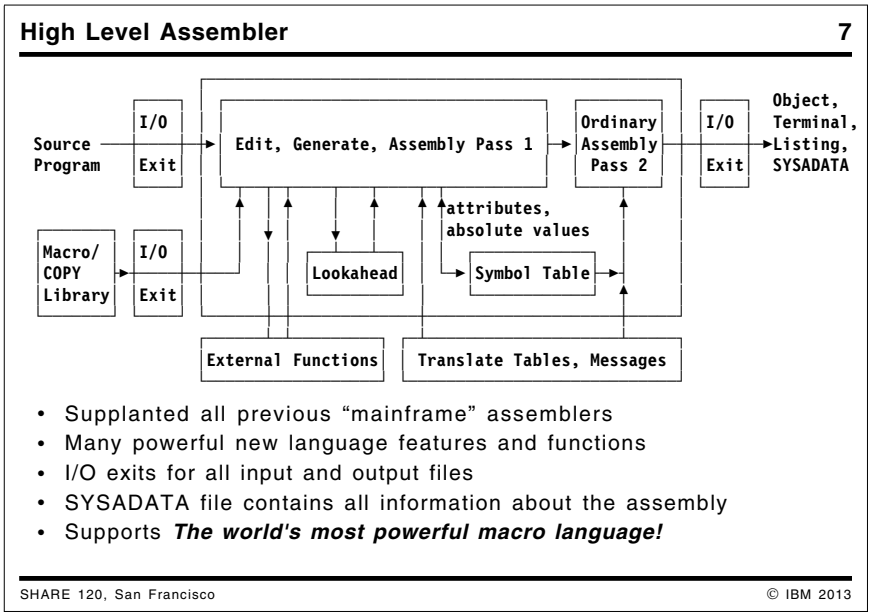
Figure 5. Dynamic Macro Redefinition

The increased power of ASMH's conditional assembly and macro language was demonstrated by its developers by writing a FORTRAN macro. The first statement of the program was FORTRAN and the remaining statements were a complete FORTRAN-language program that was read into character variable symbols, parsed, and machine instructions were generated. It helped to prove the power and value of ASMH.

ASMH Assembly Pass 2

Before the start of pass 2 in ASMH, all macro definitions are discarded to save storage. Then, an interlude iterates over the pseudo-op file and the symbol tables, connecting the disjoint internal location counters. If circular definitions occur, a diagnostic is issued and the first symbol involved receives the value of the location counter at that point. Thus, all symbols are bound before final assembly begins; during final assembly, the actual location counters and all control sections are mapped onto object-program storage.

In both ASMF and ASMH, another interlude between Passes 1 and 2 creates the External Symbol Dictionary (ESD) and issues their object module records.



High Level Assembler

Figure 6 illustrates the structure of High Level Assembler:

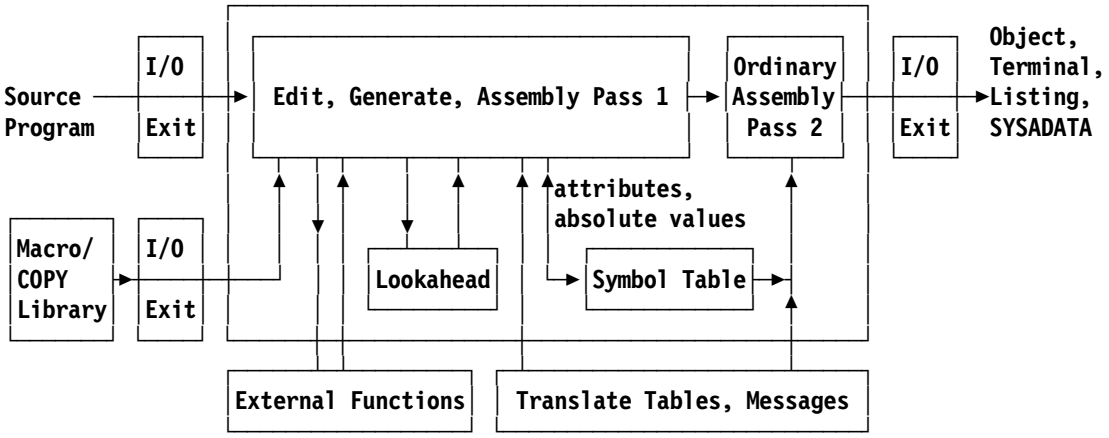


Figure 6. High Level Assembler Processing

HLASM supports an intimate interaction between the base and conditional languages that enables a macro processor with vastly superior capabilities compared to all other macro and preprocessor facilities. Other enhancements include:

- HLASM R1 (May 1992)
  - Many new options
  - Labeled, Dependent, and Labeled Dependent USINGs, active-USING headings and a USING Map
  - I/O exits to allow tailoring of all input and output
  - A SYSADATA file containing complete information about the assembly
  - Many new system variable symbols

  - Mixed-case input records
  - Four new assembler instruction statements
  - Messages in English, German, Spanish, and Japanese

- HLASM R2 (March 1995)
  - New and extensible object file format ("GOFF")
  - *PROCESS statement for module-specific options
  - Five new assembler instruction statements
  - Support for external conditional assembly functions
  - Fourteen new built-in conditional assembly functions

- HLASM Toolkit Feature (December 1995)
  - Interactive Debug Facility
  - Structured Programming Macros
  - Disassembler
  - Enhanced SuperC comparison/search utility
  - Source Cross-Reference Utility
  - Graphic Program Understanding Tool

- HLASM R3 (September 1998)
  - Support for IEEE binary floating-point data and instructions
  - Improved conversion of hexadecimal constants, with rounding options
  - AINSERT statement for conditional assembly
  - Six new system variable symbols
  - 8-byte address constants
  - 8-byte numeric binary constants
  - UNICODE™ support in character constants
  - External ASMAOPT options file
  - Many enhancements to ASMIDF and the Structured Programming Macros

- HLASM R4 (September 2000)
  - Cross-references of registers, macros, DSECTs, and unreferenced symbols
  - Support for new instructions, new and extended data types
  - Many new system variable symbols and conditional assembly internal and external functions

- HLASM R5 (June 2004)
  - Enhanced alignment facilities
  - DLL support for LE programs
  - ASCII character constants
  - Four new options
  - Program and Assembler type attributes for symbols
  - Twenty-nine new guilt-in conditional assembly functions
  - Longer conditional assembly character variables
  - Support for z File System inputs
  - ASMIDF extended to support debugging of batch programs
  - Many enhancements to Structured Programming Macros

- HLASM for Linux on zSeries (July 2008)

- HLASM R6 (July 2008)
  - HLASM Service Interface to enhance system independence
  - Decimal floating-point support for constants and instructions
  - Translation of character self-defining terms
  - Address constants supporting conditional sequential resolution