High Level Assembler:

Toolkit Feature Technical Overview

SHARE 102 (Feb. 2004), Session 8166

John R. Ehrman
ehrman@us.ibm.com or ehrman@vnet.ibm.com

IBM Silicon Valley (Santa Teresa) Lab
555 Bailey Avenue
San Jose, California 95141 USA

February, 2004

Table of Contents

Table of Contents

High Level Assembler Toolkit Feature

- Optional priced feature of High Level Assembler for MVS & VM & VSE
- Enhances productivity by providing six powerful tools:
  1. A flexible **Disassembler**
     – Creates symbolic Assembler Language source from object code
  2. A powerful Source **Cross-Reference Facility**
     – Analyzes code, summarizes symbol and macro use, locates specific tokens
  3. A workstation-based **Program Understanding Tool**
     – Provides graphic displays of control flow within and among programs
  4. A powerful and sophisticated **Interactive Debug Facility** (IDF)
     – Supports a rich set of diagnostic and display facilities and commands
  5. A complete set of **Structured Programming Macros**
     – Do, Do-While, Do-Until, If-Then-Else, Search, Case, Select, etc.
  6. A versatile **File Comparison Utility** (“**Enhanced SuperC**”)
     – Includes special date-handling capabilities
- A comprehensive tool set for Assembler Language applications

Why Use the Assembler Toolkit?

- Preserve investments in applications, people, skills, and procedures
  - Enhance the productivity of people with specialized skills
- Improve product maintainability and simplify upgrades
  - Enhancement and maintenance average 60% of software costs
- Improve application understandability
  - Product understanding typically requires 30% of maintenance time
- Improve application error detection and correction
  - Normal testing typically covers only 60% of code paths
  - Even 100% coverage can't find the 75% of defects from...
    - missing logic paths that should have been there
    - combinations of paths that aren't tested by coverage tools
- The Toolkit components can provide savings in many areas

HLASM Toolkit Publications

GC26-8709 *Toolkit Feature Interactive Debug Facility User's Guide*

The reference document for all IDF facilities, commands, windows and messages.

GC26-8710 *Toolkit Feature User's Guide*

Reference and usage information for the Disassembler, the Cross-Reference Facility, the Program Understanding Tool, the File Comparison Utility, and the Structured Programming Macros

GC26-8711 *Toolkit Feature Installation and Customization Guide*

Information needed to install all Toolkit Feature components

GC26-8712 *Toolkit Feature Interactive Debug Facility Reference Summary*

Quick-reference summary, with syntax of all commands and a list of all options; for experienced ASMIDF users.

HLASM Toolkit Disassembler

- Converts object code to Assembler Language source
- Supports latest processor instructions, including z/Architecture
- Input files:
  - Object modules; MVS load modules and program objects; CMS modules; VSE phases
  - Control statements (including a COPYLIB)
- Output files:

  LISTING control records, messages, source listing, etc.

  PUNCH assembler-ready source file, to re-create the object
- Limitations:
  - 16MB upper limit on size of module being disassembled
  - MVS: no Program Objects containing nonstandard classes
  - No Generalized Object File Format (GOFF) object files
  - VSE: phases have no ESD; cannot extract individual CSECTs
  - SYM-record information not used, even if present

Disassembler Operation

- Copyright protection and the `COPYRIGHTOK` option
- Control statements add symbolic and structure information

  DATA, INSTR, DS
  designate data, code, and empty areas

  DSECT provides symbolic mappings of structures

  ULABL assigns user labels to points in the program

  USING provides basing data to allow symbolic references in place of explicit base-displacement operands

  COPY includes previously created control statements

- Symbolic names automatically provided for <u>all</u> registers
  - Access, Control, Floating-Point, General Purpose, and Vector
- Informative comments on SVCs, STM, EX, BAL, BALR, etc.
- Listing contains ESD, RLD, other useful information

Disassembler Usage

- Initial disassembly
  - Specify the module and CSECT to be disassembled
- Add USING records
  - Specify base registers, contents, and USING ranges
- Add other control records
  - Specify areas used for instructions, data, and “empty space”
  - Assign your own labels to known instructions, data areas, work areas
  - Map data structures with DSECT statements
- Program Understanding Tool helps clarify structure
  - Especially useful for compiled HLL code
- Place control records in separate files, include COPY statements

HLASM Toolkit Cross-Reference Facility

- Scans source, macros, and COPY files for
  - symbols, macros, and user-specified character strings (“tokens”)
- Full support for Assembler, C/C++, PL/I, REXX
  - Extensive support for many other languages, including COBOL, FORTRAN, JCL, CLIST, ISPF, RPG, SCRIPT, SQL, PL/X, etc.
- Can create a source file with token matches “tagged”
  - Useful as input to Program Understanding tool
- Recent enhancement! APAR PQ67403 adds:
  - 31-bit enablement for larger reports
  - New SYMC “compact symbol-sort-order” for SWU reports
  - Message limits now apply independently to each severity

HLASM Toolkit Cross-Reference Facility ...

- Produces up to six reports
  - Control Flow (CF)
  - Lines of Code (LOC)
    - Lines of OO code (LOOC) for C/C++
  - Macro-Where-Used (MWU)
  - Symbol-Where-Used (SWU) (compact or expanded format)
  - Token-Where-Used (TWU)
    - Supports generic (wild-character) matching, “exclusion” tokens
  - Spreadsheet-Oriented (SOR)
    - Same info as TWU, but in a format useful for identifying critical modules and estimating conversion effort
- Can create a source file with token matches “tagged”
  - Useful as input to Program Understanding tool

HLASM Toolkit Program Understanding Tool

- Detailed analysis of Assembler Language programs
  - Creates annotated listings
  - Displays graphic control flow for single programs and “linked” modules
  - Runs on Windows and OS/2
- Assemble programs with ADATA option
  - Download SYSADATA file (in binary) to workstation `*.XAA` files
- ASMPUT analyzes the SYSADATA (`.XAA`) files
  - Creates component lists, simulated listing, graphs, external linkages
- Grapher displays many levels of detail, with zoom capability
  - Inter-program relationships
  - Major program structures
  - Full details of internal control flows
  - Graph-printing test version available on HLASM web site
- Online tutorial, extensive HELPs throughout
  - Windows Help requires Internet Explorer
- Installed from downloaded host files (not diskettes)

HLASM Toolkit Interactive Debug Facility (IDF)

- Supports latest processor enhancements
  - 64-bit instructions and AMODE(64)
    - APAR PQ51325, Requires HLASM R4 and z/OS 1.2 or later
    - New options, commands, and windows
  - additional floating point registers and new FP instructions
- Primarily for Assembler Language programs
  - Also usable for programs in other languages
    - Without source-language support
- Multiple selectable “windows” for address stops, breakpoints, register displays, disassembled code, register histories, etc.
  - Windows may be used in any order or combination

HLASM Toolkit Interactive Debug Facility (IDF) ...

- Execution stepping: displays disassembled code (and source, if available)
  - Per instruction, or between breakpoints or routines
  - Breakpoints include “watchpoints” (break on specified condition)
  - Instruction counting, execution “history”
- Exit routines (in REXX or other language) invokable at breakpoints
  - Capture, analyze, and respond to program conditions
- Storage and register modification by over-typing
- Record/playback facility to re-execute debugging sessions
- Extensive tailoring capabilities
- GC26-8709**-04**, *High Level Assembler Toolkit Interactive Debug Facility User's Guide* (*Reference Summary* is GC26-8712**-03**)
  - 64-bit debug info is available in soft-copy only

Interactive Debug Facility (IDF) Overview

- Components
  - Base Debugger: ASMIDF can be used without source-language support
    - On CMS, includes interface module
  - ASMLANGX (Extraction Utility) prepares HLASM ADATA files
- Two breakpoint types: SVC97, invalid opcodes (X'01xx')
- System considerations
  - TSO: naming conventions; etc.
    - Supports DFSMS/MVS Binder Program Objects (standard classes)
    - SVC97 option if application uses ESPIE/ESTAE; subtask of IDF
    - NOSVC97 option if application uses TSO TEST; same task as IDF
  - CMS: Invalid opcodes only (NOSVC97); PER support
  - VSE: Link with ASMLKEDT, specify VTAM terminal
  - ISPF: TSOEXEC command (IDF “owns” the screen)
  - CICS, DB2, IMS with some limitations
  - Debugging authorized code: not supported!
  - LE: specify `NOSPIE`, `NOSTAE` (or `TRAP(OFF)`)

ASMIDF: Preparing a Debug Session

- Without source level facilities
  - On CMS: LOAD MAP file required
  - On VSE: link edit with ASMLKEDT
- With source level facilities
  1. Assemble with High Level Assembler's `ADATA` option
  2. Run ASMLANGX extraction program against SYSADATA file
     - Prepares source and symbolic information for debug use
     - Recent APAR PQ61239 enhances performance
  3. Keep the ASMLANGX extraction file
     - Can generate the file on TSO, CMS, or VSE, and ship to the others
  4. Create target module from object file(s)
     - Require LOAD MAP file on CMS; `phasename.MAP` on VSE
     - No need to retain listing or SYSADATA files

ASMIDF: Invocation

- Invocation options vs. dynamic options
  - Almost all options may be changed dynamically
- Plan for storage utilization by applications and IDF
- Basic syntax for invoking IDF:

```
ASMIDF <module> (<ASMIDF options> / <module parameters and options>
```

  - Example: debugging HLASM's CMS interface module:

```
ASMIDF  ASMAHL  ( AMODE31 NOPROF  / TESTASM (SIZE(1M)
```

- IDF gains control on program checks, ABENDs, breakpoints, program completion, break-in interrupts, etc.
- Trace dynamically-loaded modules with deferred breakpoints

```
DBREAK (loaded_module.csect_name)
```

- ISPF invocation: Under option 6, use **TSOEXEC** command

ASMIDF: Useful Options

PROFILE/NOPROFIL

IDF by default looks for PROFILE ASM (a REXX exec)

AMODE24/AMODE31/AMODE64

Sets initial AMODE of target program

AUTOSIZE/NOAUTOSZ

Controls automatic window resizing

PATH, FASTPATH

Counts number of instruction executions

LIBE Specifies library containing target application module

CMDLOG, RLOG

Create or append to or replay command log file

ASMIDF: Debugger Windows

- Command Window (always displayed)
- Current Registers: General (32 or 64 bit), Access, Control, Float
  - APFR for 16 Floating-Point registers
- Old Registers
- Break (breakpoints and watchpoints)
- Disassembly (multiple)
- Dump (multiple)
- Entry Point Names
- Language Support Module Information
- Minimized Window Viewer
- Options
- Skipped Subroutines
- Target Status
- ADSTOPS (CMS only: uses PER; supports REGSTOPS also)

ASMIDF: Useful Debugger Commands

- **BREAK**: Set a breakpoint, or display the Break Window
- **DBREAK**: Set a deferred (“sticky”) breakpoint
- **DUMP**: Display storage in symbolic or “dump” format
- **FIND/LOCATE**: Locate and display given strings in storage
- **HISTORY**: Display previously executed instructions
- **WATCH**: Specify a break-test condition at a “watchpoint”
- **DISASM**: Disassemble a specified area of storage
- **STEP/STMTSTEP/RUN**: Control instruction-execution rates
- **FOLLOW**: Dynamically track contents of a register or word in storage
- **LANGUAGE LOAD**: Load specified language-extraction files
- **HIDE/SHOW**: Control display detail of source and disassembly data
- **UNTIL**: Execute instructions up to a specified address
- ...nearly 190, in all!
- New, for 64-bit debugging: **REGS64, GPRG, GPRH, EPNAMES**

ASMIDF: Debugger Macros

- REXX (interpreted or compiled)
  - A very powerful extension mechanism
- Default address
- **EXTRACT** command (almost 90 different items available to macros)
- **IMPMacro** option for automatic macro search (ON by default)
- **MRUN/MSTEP** commands to control execution from macros
- **PROFILE** macro to customize your environment
- **EXIT** routine may gain control at specified events

ASMIDF: Debugger Macros, Example 1

```
/*==================================================================\
| TRAP macro:  uses DBREAK to load and break on the entry point of  |
|              a loadable module                                    |
| PARAMETERS:  name - module name                                   |
|              symbol - external symbol to set break point on       |
\==================================================================*/

arg name symbol .
if name == '' then exit 99
if symbol == '' then symbol = name
'DBREAK ('name'.'symbol')'  /* Issue DBREAK at start of CSECT    */
'MRUN'               /* Program will run until DBREAK is matched */
'QUAL' name                 /* Change qualifier                  */
'LAN LOAD' symbol           /* Load extraction file              */
'BREAK' symbol              /* Remove breakpoint at module start */
exit
```

ASMIDF: Debugger Macros, Example 2

```
/*REXX ───────────────────────────────────────────────────────────*/
/*            REGS – Toggle the current registers window.         */
/*                                                                */
/* When the REGS window is opened, it will be moved on the ASMIDF */
/* display so that it is the first window.                        */
/*────────────────────────────────────────────────────────────────*/

 'REGS'                           /* Toggle REGS window           */

 'Extract Cursor'                 /* Obtain window information    */
 n = Find(display,'REGS')         /* Is REGS window present?      */
 If n ¬= 0 Then                   /* Yes?  Force to be 1st window */
   'ORDER ='n

 Exit
```

HLASM Toolkit Structured Programming Macros

- Macro sets can help eliminate test/branch instructions, simplify program structures:
  1. **If-Then-Else**, **If-Then** (`IF/ELSEIF/ELSE/ENDIF`)
  2. **Do, Do-While, Do-Until** (`DO/ITERATE/DOEXIT/ASMLEAVE/ENDDO`)
     - supports forward/backward indexing, FROM-TO-BY values, etc.
  3. **Search** (`STRTSRCH/ORELSE/ENDLOOP/ENDSRCH/EXITIF`)
     - supports flexible and powerful choices of loop controls and test conditions
  4. **Case** (`CASENTRY/CASE/ENDCASE`).
     - provides rapid switching via N-way branch to specified cases
  5. **Select** (`SELECT/WHEN/OTHRWISE/ENDSEL`) with two forms!
     - allows general choices among cases using sequential tests
- All macro sets may be (properly) nested in any order, to any level
- You can use the full instruction set (including the newest ops)

Structured Programming Macros: Why Use Them?

Many users report the following benefits:

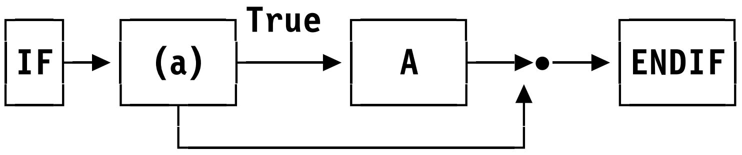
- Improved code readability and understandability
- Faster application development
- Cleaner code
- Eliminating extraneous labels makes code easier to revise
- You can use the SP macros when and where appropriate
  - Introduce the macros incrementally
- APAR PQ69812 adds extensive generalizations and improvements
  - APAR PQ74641 changes LEAVE to ASMLEAVE (IMS problem) and allows easy renaming of any macro

Structured Programming Macros: Usage

- All macros are contained in a single member, ASMMSP
  - Use `COPY ASMMSP` statement to initialize
  - Or specify `PROFILE(ASMMSP)` option
  - Packaging dictated by IBM naming rules/conventions
- User macros have meaningful mnemonics
  - Internal (non-user) macro names begin with `ASMM`
- Global variables now begin with `&ASMA_` to prevent conflicts
- GC26-8710, *High Level Assembler Toolkit User's Guide*

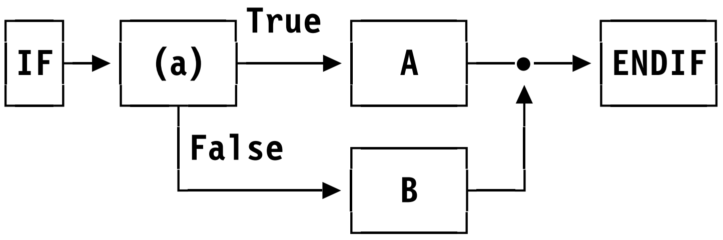
Structured Programming Macros: IF-THEN-ELSE

- Basic IF-ENDIF:



```
IF (a) Then
  A
ENDIF
```

- Basic IF-ELSE-ENDIF:



```
IF (a) Then
  A
ELSE
  B
ENDIF
```

- The word THEN is **not** syntactic; only a comment
  - Used only to improve readability, understandability

Structured Programming Macros: Example 1

- Add absolute value of c(R4) to c(R5); don't change R4
- Unstructured:

```
       LTR   R4,R4                  Set CC
       BM    LABEL1                 Negative? Branch
       AR    R5,R4                  Positive or zero – add to R5
       B     LABEL2                 Skip the negative case
LABEL1 DS   OH
       SR    R5,R4                  Subtract negative value
LABEL2 DS   OH
```

- Structured:

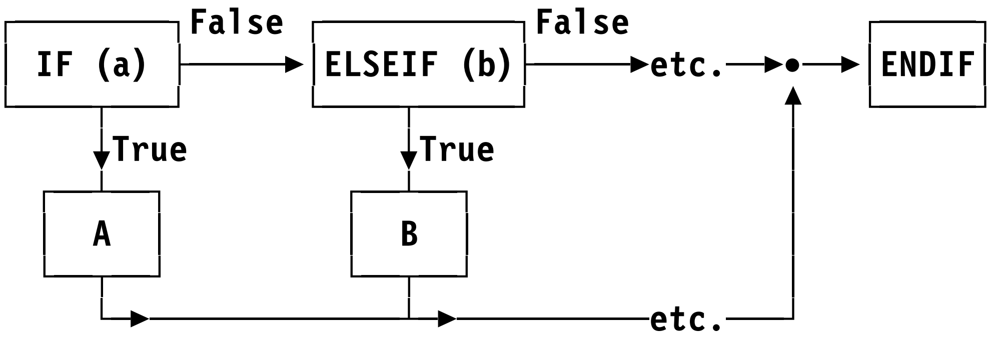
```
       IF    (LTR,R4,R4,NM)  THEN   Test R4 for non–negative
         AR    R5,R4                Positive or zero – add to R5
       ELSE  ,                      Otherwise,
         SR    R5,R4                Subtract negative value
       ENDIF
```

- Can also use relative-immediate instructions:

```
       IF    (CHI,15,EQ,–3)         Compare with Halfword–Immediate
```

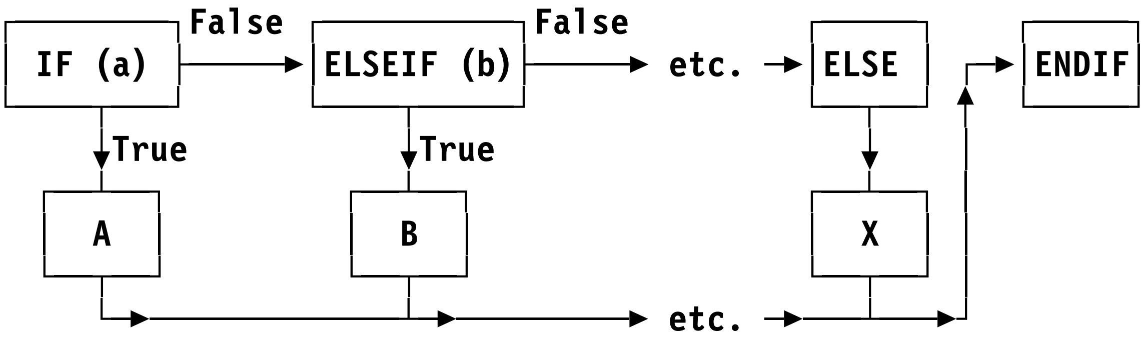
Structured Programming Macros: IF-THEN-ELSEIF-ELSE

- The ELSEIF macro simplifies deep nesting of IF-ELSE-ENDIF groups:



```
IF (a) Then
  A
ELSEIF (b)
  B
– – –
ENDIF
```

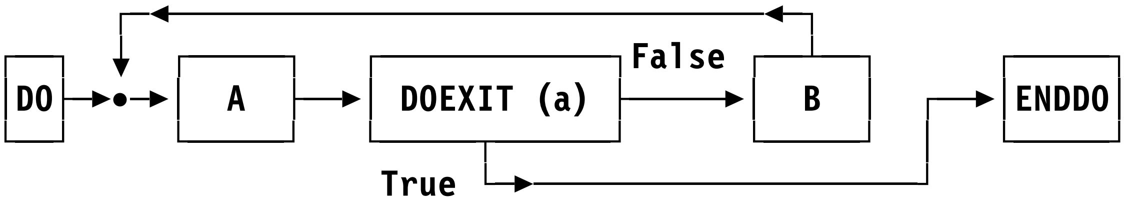
- Also used with an ELSE clause:



```
IF  (a) Then
  A
ELSEIF (b)
  B
– – –
ELSE
  X
ENDIF
```

Structured Programming Macros: DO Set

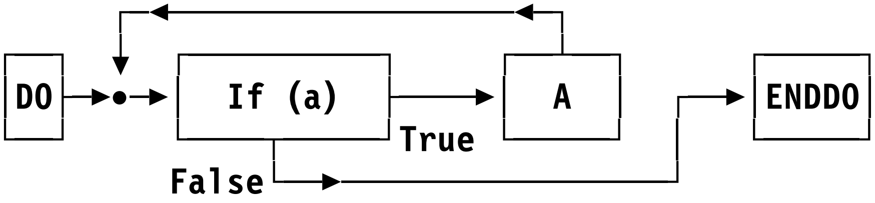
- **DO, DO-WHILE, DO-UNTIL** `predicates` support mixtures of WHILE, UNTIL, forward/backward indexing, FROM-TO-BY values, etc.
  - DOEXIT macro uses IF-macro syntax to exit the containing DO
  - ASMLEAVE exits any number of containing labeled DOs
  - ITERATE requests immediate execution of the next loop iteration for any containing DO
- A *very* rich and flexible set of facilities
- Simplest form: infinite loop, exited with a DOEXIT macro



```
DO INF
  A
DOEXIT (a)
  B
ENDDO
```

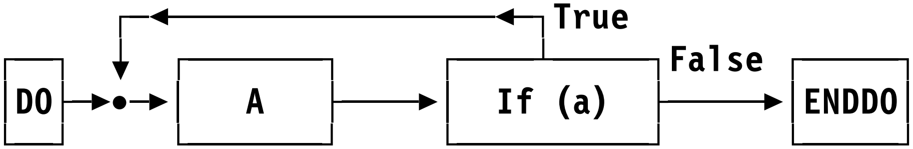
DO-WHILE and DO-UNTIL

- DO-WHILE tests before entering a loop:



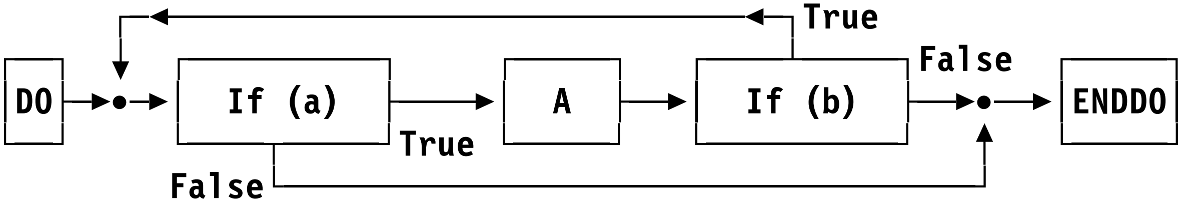
```
DO WHILE=(a)
  A
ENDDO
```

- DO-UNTIL tests after executing a loop:



```
DO UNTIL=(a)
  A
ENDDO
```

- DO-WHILE and DO-UNTIL tests can be combined:



```
DO  WHILE=(a),UNTIL=(b)
  A
ENDDO
```

```
DO  UNTIL=(b),WHILE=(a)
  A
ENDDO
```

Structured Programming Macros: Example 2

- Search a string for first blank character, or end of string
- Unstructured:

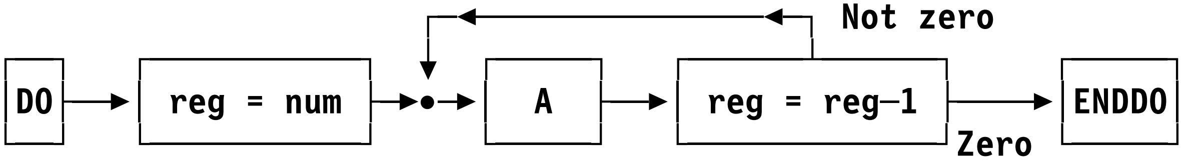
```
       L     R5,=A(Start–1)      Address start–1 of expression
Top_of_Loop DS 0H
       C     R5,End              Test for end of expression
       BNL   Leave_Loop          and exit if we've reached end
       LA    R5,1(,R5)           Move along one byte
       CLI   0(R5),C' '          Test for a blank
       BNE   Top_of_Loop         not yet, repeat loop
Leave_Loop  DS 0H
```

- Structured:

```
       L     R5,=A(Start–1)      Address start–1 of expression
Scan   DO WHILE=(C,R5,LT,End),UNTIL=(CLI,0(R5),EQ,C' ')
         LA    R5,1(,R5)         Move along one byte
       ENDDO
```

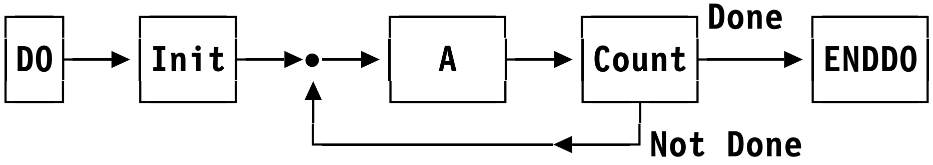
Structured Programming Macros: Iterative-Do Macros

- Two styles: simple count, general indexing
- Count style does a set number of iterations



```
DO FROM=(reg,num)
  A
ENDDO
```

- Indexing form is extremely flexible

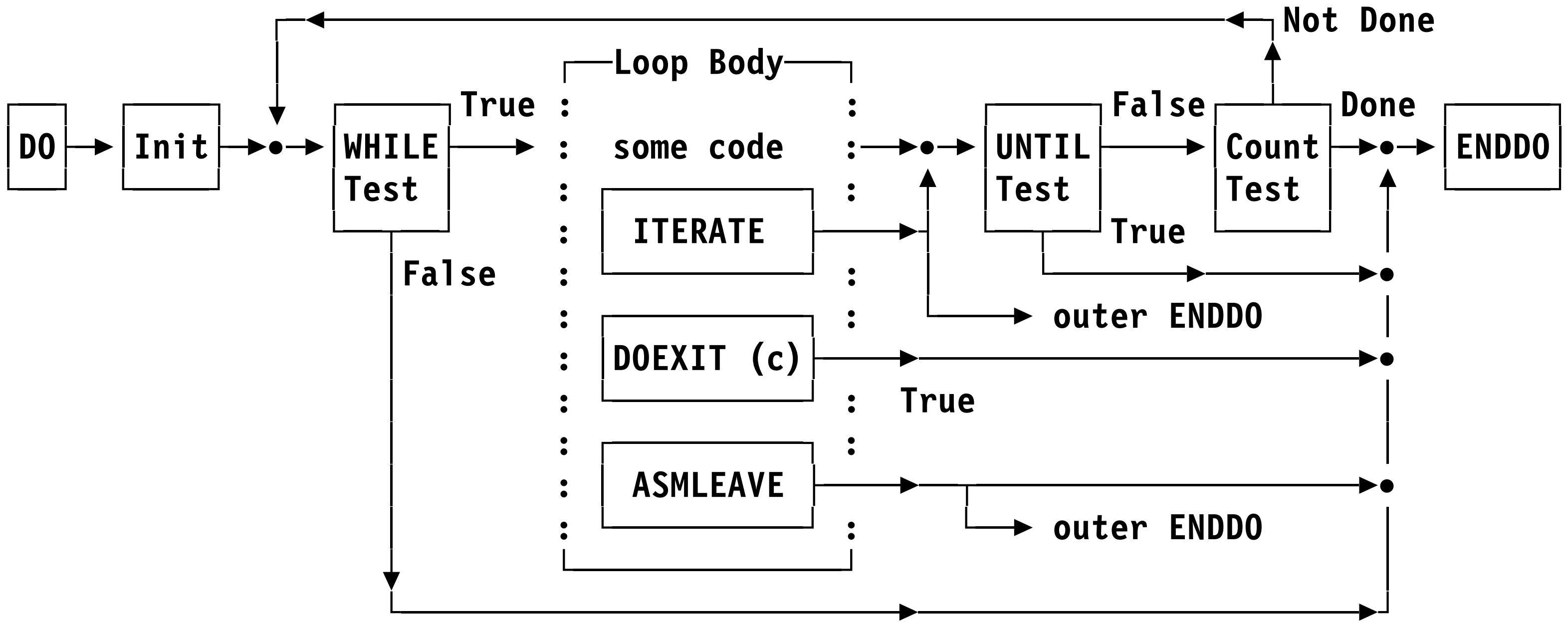


```
DO  [BXH|BXLE,]FROM=(Rx,num),TO=(Ry+1,num),BY=(Ry,num)
  A
ENDDO
```

  - Counts up or down
  - Automatic or user selection of BXH or BXLE loop closing
  - Many variations supported

Structured Programming Macros: General Form of DO Statement

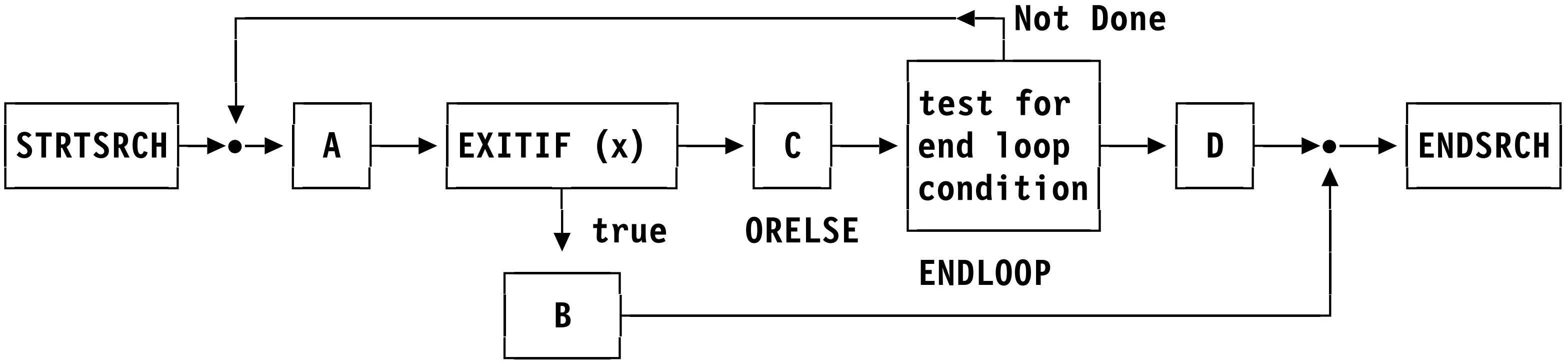
- DO statement supports a rich combination of operands



- You can specify very detailed loop controls

Structured Programming Macros: SEARCH Set

- SEARCH set specifies a complex looping structure:



- Statement format:

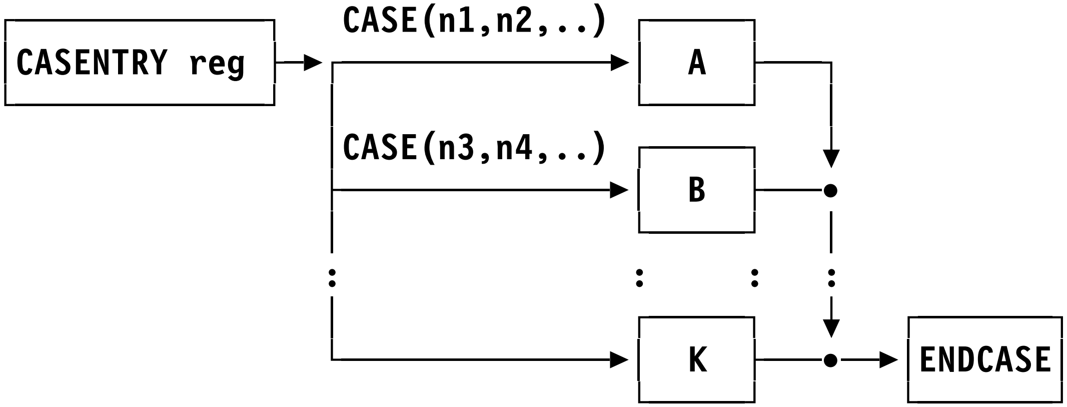
```
STRTSRCH  (any DO–loop operands)
  Process Code A
EXITIF    (any IF–type operands)  ┐
  Process Code B                  │ repeatable
ORELSE            ┐ last one      │ clauses
  Process Code C  ┘ optional      ┘
ENDLOOP
  Process Code D
ENDSRCH
```

Structured Programming Macros: CASE Set

- CASE macros provide rapid selection of blocks of code

```
CASENTRY  register[,POWER=p,VECTOR=B|BR]
 CASE   n1,n2,...
   Process Code A
 CASE   n3,n4,...
   Process Code B
 _ _ _ _
ENDCASE
```

- register operand contains an integer power of 2, **p**
- VECTOR operand selects table of branches, or adcons used by BR



Structured Programming Macros: SELECT Set

- SELECT group with single comparison:

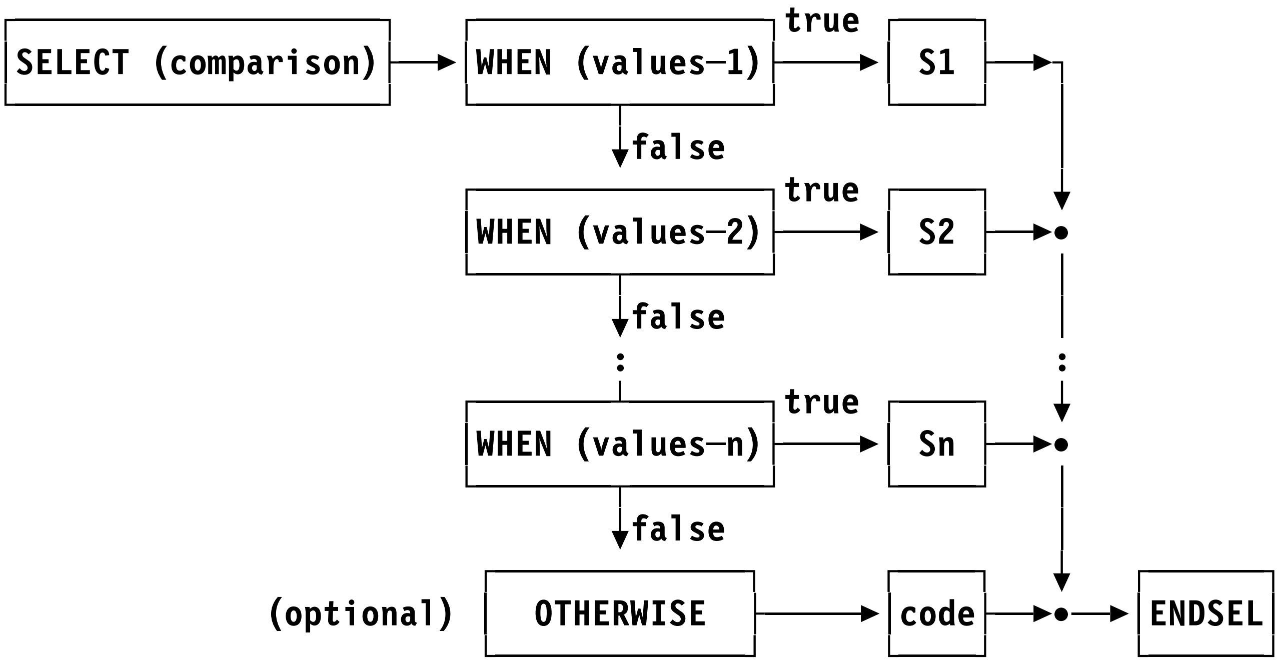
```
SELECT (comparison)            Compare instruction & condition
  WHEN   (list-of-values-1)      Values for this comparison
    <statements-1>               Statements for these cases
• • •
  WHEN   (list-of-values-n)      Values for last comparison
    <statements-n>               Statements for these cases
  OTHRWISE ,
    <statements>                 Executed if no matching WHEN
ENDSEL ,                       End of SELECT group
```

- SELECT group with multiple comparisons/tests:

```
SELECT ,                       No operands
  WHEN   (comparisons-1)         Comparisons and/or tests
    <statements-1>               Statements for these cases
• • •
  WHEN   (comparisons-n)         Comparisons and/or tests
    <statements-n>               Statements for these cases
  OTHRWISE ,
    <statements>                 Executed if no matching WHEN
ENDSEL ,                       End of SELECT group
```

Structured Programming Macros: Single-Comparison SELECT

- Same comparison used for all WHEN clauses



- WHEN operand is a list of one or more items
- Easy way to test a series of identical data types

Structured Programming Macros: Single-Comparison SELECT ...

- Example: check for characters in arithmetic expressions

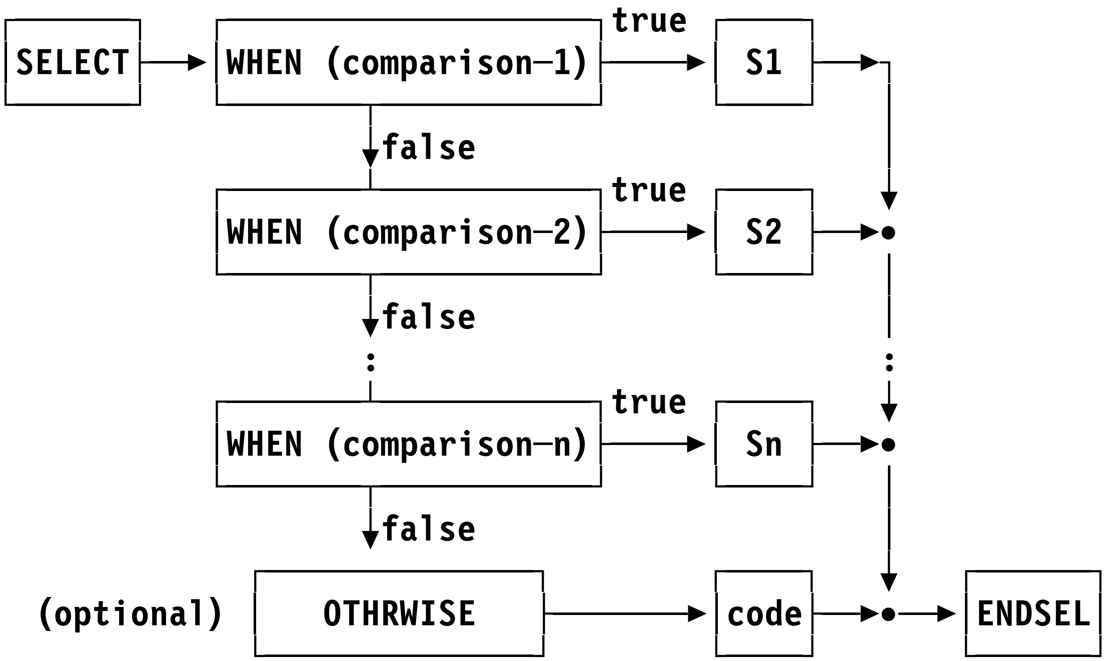
```
SELECT   (CLI,Flag,eq)
  When   (C'*',C'/',C'+',C'–')
    S1
  When   (C'(',C')',C'=')
    S2
  OTHRWISE
    code
ENDSEL
```

- Example: test small numbers in R1 for primes

```
SELECT  C,R1,Eq
  WHEN  =F'0'
    ErrorMsg 'Zero not a valid prime'
  WHEN  (=F'1',=F'2',=F'3',=F'5',=F'7')
    MVI  Flag,Prime
  WHEN  (=F'4',=F'6',=F'8')
    MVI  Flag,NotPrime
  OTHRWISE
    MVI  Flag,Unknown
ENDSEL
```

Structured Programming Macros: Multiple-Comparison SELECT

- Different comparisons/tests on each WHEN clause



- WHEN operands may be very complex
- Easy way to select among alternatives involving different types

Structured Programming Macros: Multiple-Comparison SELECT ...

- Example using mixed comparisons

```
SELECT
  When   (CLI,Flag,eq,C'+'),Or,(CLI,Flag,eq,C' ')
    S1
  When   (CLI,Flag,eq,C'–'),And,(LTR,R0,R0,M)
    S2
  – – –
  OTHRWISE
    code
ENDSEL
```

- Example: test value in R1 for a small prime

```
ST    R1,Temp
SELECT
  When   (LTR,R1,R1,P),And,(C,R1,lt,=F'4')
    MVI  Flag,Prime        R1 contains 1, 2, or 3
  When   (TM,Temp,NZ,2)    Is it even?
    MVI  Flag,NotPrime
  OTHRWISE
    MVI  Flag,UnKnown
ENDSEL
```

Structured Programming Macros: Detailed Example

- An elaborate example is provided in the text
  - Illustrates all of the macros, and all their options
  - Nested in various combinations

Source See Appendix A, “Sample structured macro program”

Listing See Appendix B, “Listing of sample program”

Structured Programming Macros: Notes

- To generate relative branches, code ASMMREL ON (OFF for based)
  - Base register not required for generated code!
- Be **very** careful about continuations! (Run with FLAG(CONT) option)
- Boolean expressions partially optimized
  - Evaluated only as far as necessary to determine result
  - Can sometimes be simplified: NOT (A AND B) = ((NOT A) OR (NOT B))
- Limitation to at most 50 operands on any one macro
  - Parentheses in operands are optional, but helpful
- Some macro operand “keys” not safely usable as program symbols:

```
P, M, O, Z, H, L, E, NP, NM, NO, NZ, NH, NL, NE,
GT, LE, EQ, LT, GE, AND, OR, ANDIF, ORIF
```

- IF, DOEXIT, EXITIF, WHEN macros allow CC= as only operand
- Don't forget the ENDxxx macros!

HLASM Toolkit Feature File Comparison Utility

- File Comparison Utility (“Enhanced SuperC”)
  - A powerful and general file comparison and search utility for individual files, or multiple libraries
  - Batch mode on MVS and VSE; panel or command line on CMS
- Compares entire files, or individual lines, words, or bytes
  - File types include load modules, VSAM ESDS+KSDS
  - Include and exclude selected data types, lines, columns, rows, etc.
- Search facility supports multiple search strings, in specified columns
  - Search strings may be words, prefixes, or suffixes
  - Multiple strings may be forced to match only on single lines
- Date-management support includes
  - Fixed or sliding windows
  - Multiple date formats and representations
  - Automatic “aging” of specified date fields
- Recent enhancements: 31-bit support (APAR PQ66218); FINDALL option (APAR PQ51367)

HLASM Toolkit Feature Usage Scenarios

1. ***Recovery*** from object/load modules (if original source is lost)

   - **Disassembler** initially produces “raw” Assembler source from “binary”
   - Control statements define code, data, USINGs, labels, DSECTs, etc.
   - Repeat disassembly/analysis/description/assembly cycle until satisfied

2. ***Analysis and understanding*** of Assembler Language source programs

   a. **ASMXREF** cross-reference token scanner

   - Locates important symbols, user-selected “tokens”
   - Creates “impact-analysis” spreadsheet-input file for effort estimation

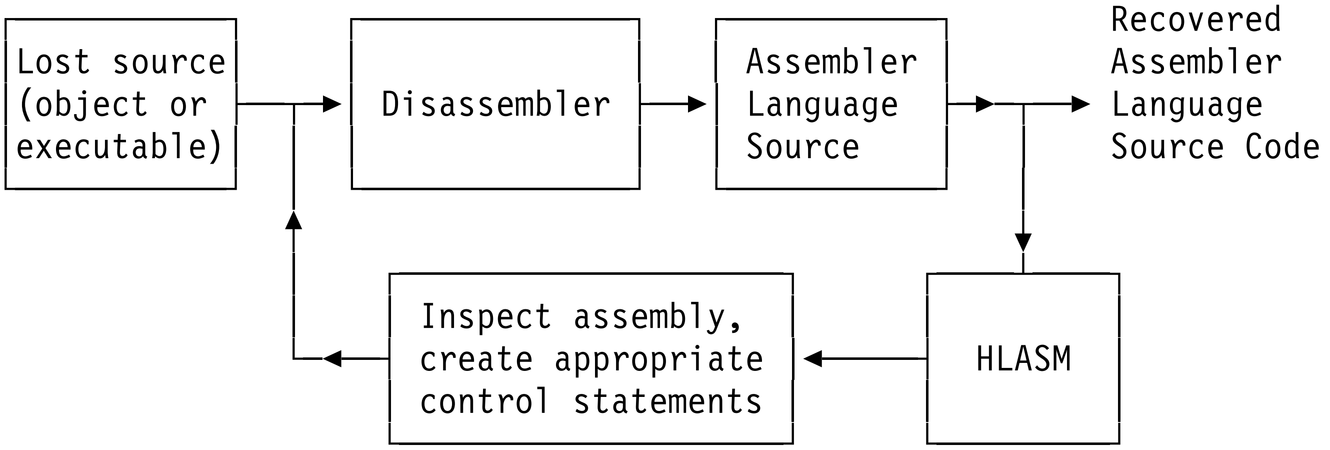
   b. **ASMPUT** Program Understanding tool

   - Graphic displays of program structure, control flow, with any level of detail
   - Can be used to ***help*** reconstruct (lost) source in HLLs!

3. ***Modification, testing, and validation*** of updated programs

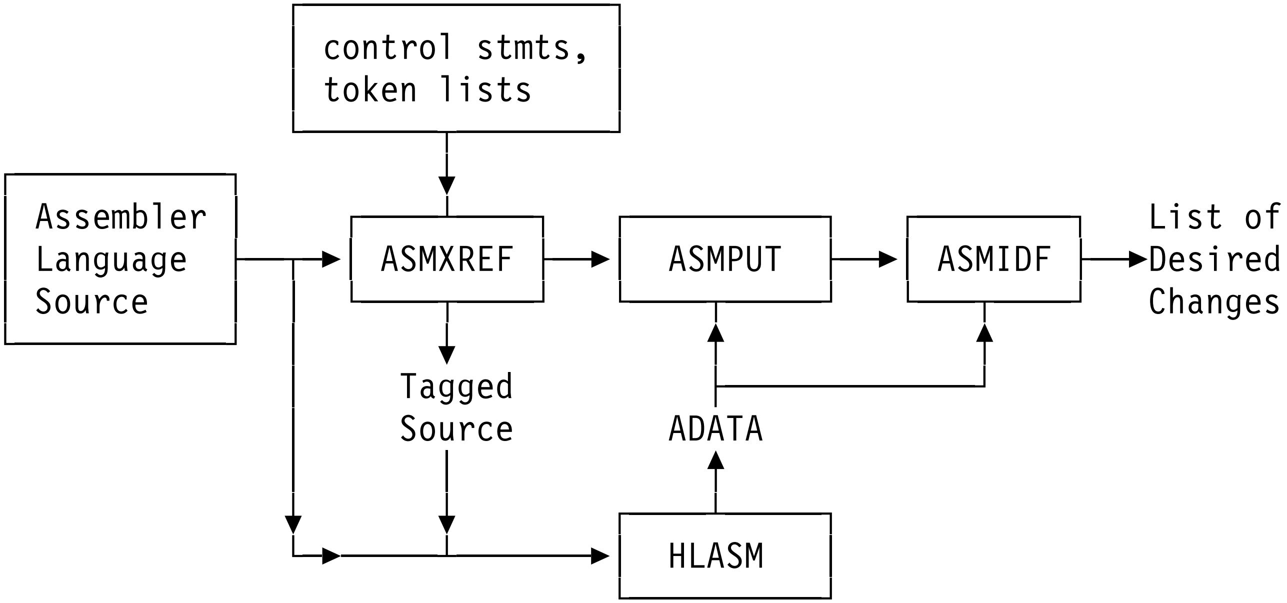
   - **Interactive Debug Facility** speeds and simplifies program testing
   - **Structured Programming Macros** clarify program coding logic
   - **File Comparison Utility** tracks before/after status of source, outputs

HLASM Toolkit Feature: Recovery and Reconstruction


Lost source (object or executable)
Disassembler
Assembler Language Source
Recovered Assembler Language Source Code
HLASM
Inspect assembly, create appropriate control statements

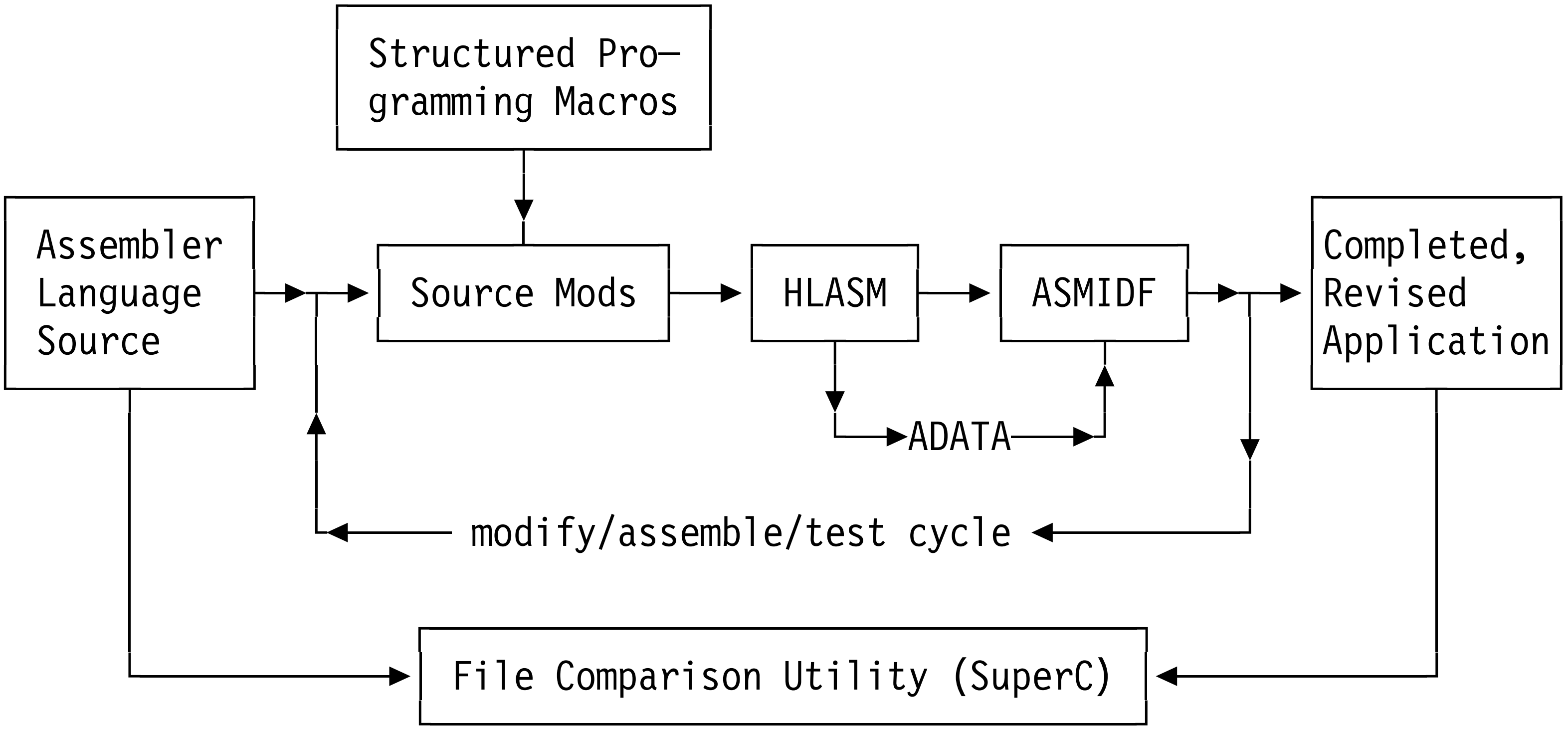
- Start with object code (object files or executables)
- Disassemble and inspect; create control statements to describe the program more fully
- Repeat this cycle as more of the program is understood
- Readable source is used as input to later phases

HLASM Toolkit Feature: Analysis and Understanding



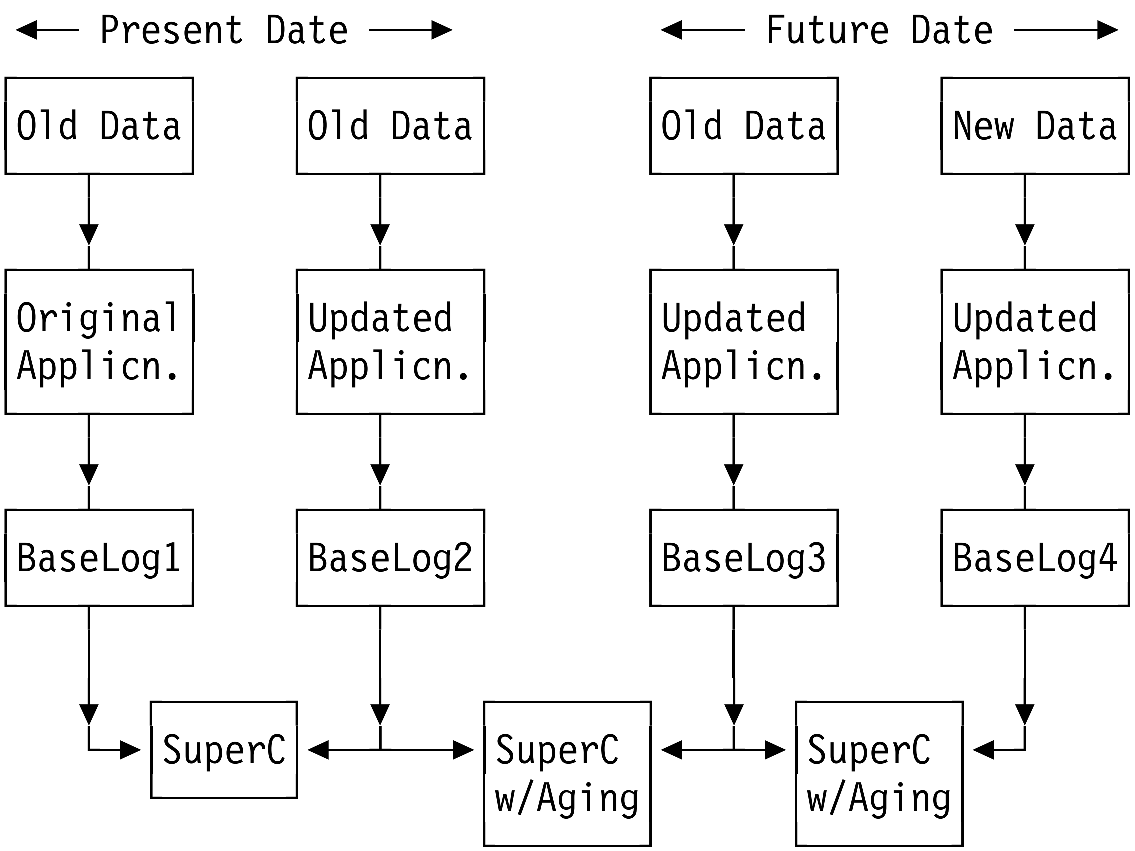
- ASMXREF scans assembler source programs, identifies key items
  - Create “tagged” source file identifying important “tokens”
- Assemble; ASMPUT uses ADATA to analyze control flows
- Use IDF to trace data flows in detail

HLASM Toolkit Feature: Modification and Testing



- Modify Assembler Language source at desired points
- Assemble and execute the program, test with IDF
- Make indicated modifications until result is satisfactory
- Compare original and updated source files to validate changes

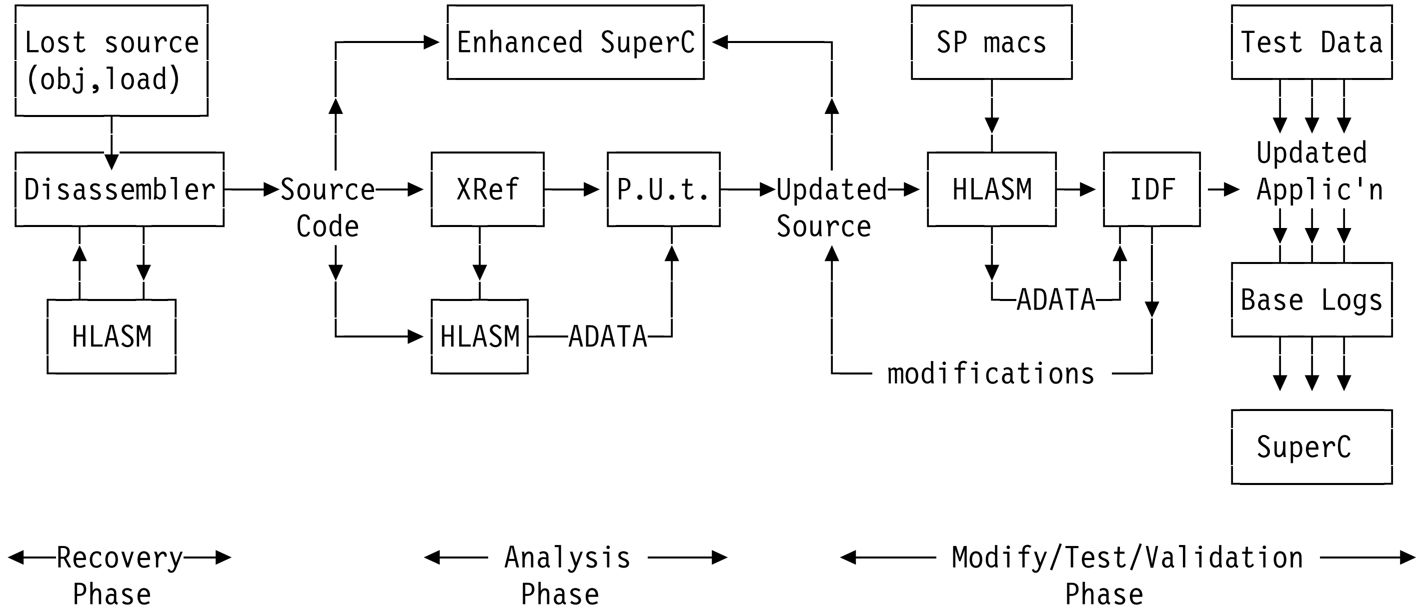
HLASM Toolkit Feature: Validation



- Create “base logs” with original and updated application, current and “future” dates, and old and modified data
- Compare results at each stage using “Aging” facilities as needed

HLASM Toolkit Feature: Scenario Summary

- The Toolkit Feature's components support all phases of Assembler Language development, maintenance, and migration:



HLASM Toolkit Feature: Full-Spectrum Application Support

Activity	Toolkit Feature Components
Inventory, assessment	**Disassembler** helps recover programs
Locating key fields	**Cross-Reference Facility** pinpoints named fields, localizes references **File Comparison Utility** searches files for strings
Application understanding	**Program Understanding Tool** provides insights into program structures and control flows; **Interactive Debug Facility** monitors instruction and data flows at any level of detail
Decide on fixes	**...**
Implement changes	**Structured Programming Macros** clarify source code; **Enhanced SuperC** helps validate source changes
Unit test	**Interactive Debug Facility** provides powerful debugging and tracing capabilities
Debug	**Interactive Debug Facility** debugs complete applications, including loaded modules
Validation	**Enhanced SuperC** checks regressions, validates correctness of updates

HLASM Toolkit: Summary

- HLASM Toolkit Feature provides a powerful, flexible toolset:
  1. Disassembler
  2. Cross-Reference Facility
  3. Program Understanding Tool
  4. Interactive Debug Facility
  5. Structured Programming Macros
  6. File Comparison Utility (Enhanced SuperC)
- Supports almost all development and maintenance tasks
  - On OS/390, MVS/ESA, VM/ESA, and VSE/ESA
- HLASM web site: demos of ASMPUT, ASMIDF (basic and advanced); 30-day free trial version of ASMPUT