Upgrading and Modernizing Your Assembler Language Programs: Tips, Techniques, and Technologies

SHARE 120 in San Francisco, Session 12522

John R. Ehrman
ehrman@us.ibm.com

IBM Silicon Valley Laboratory
555 Bailey Avenue
San Jose, CA 95141

February 2013

Topics

- Introduction: Why modernize? What are the alternatives?
- Part 1: Tidying up portions of your programs – 10 simple examples
- Part 2: Improving readability and maintainability
  - Ways you can clarify and simplify program organization
  - Easy changes that can make small segments of code more manageable
- Part 3: Upgrading instructions to newer, efficient forms
  - Simple instructions that can make code clearer, smaller, and more efficient
- Part 4: Performance sensitivity: enhancing awareness of CPU behavior
  - Little things that can make critical sequences more efficient
- Summary observations

Why Modernize?

What should be done with old Assembler Language Programs?

Some possibilities:

1. Live with what you have, modify as needed
2. Replace existing applications with a commercial package
3. Use automated conversion tools to convert the assembler code to a "modern" language
4. Carefully document the existing code, then rewrite it in a new language
5. Modernize and upgrade the existing applications

Each choice has considerations for and against. Research has shown:

- Changing languages has many hidden costs, and should be avoided.
- Problem-domain expertise is often more important than programming-language expertise
- High-level languages do not provide improved reliability or maintainability

Things to consider for each possibility

Option	Factors
Keep what you have	• Simplest, cheapest choice • Possible lost business opportunity
Buy a package	• Specifying, negotiating requirements may be lengthy • Transition and testing require system stability • Possible hidden costs of tailoring, training, maintenance
Automated conversion to a HLL	• Rarely successful; technically difficult to validate • Code freeze; possible lost business opportunity • Converted applications *and* test cases often require modification • Bilingual staff skills required • Converted programs typically slower and fatter
Rewrite in a new language	• Code freeze; possible lost business opportunity • Rewrite applications *and* test cases • Large testing costs, validating test cases themselves • Bilingual staff skills required
Modernize, upgrade	• Incremental, no cut-overs; stable code base • New techniques easy to apply • No lost business opportunity • Typically significant reduction in maintenance costs

Part 1: Easy ways to “tidy” and simplify your programs

Assembler Language doesn't *have* to be difficult!

Example 1: Initializing a buffer

- A common instruction sequence

```
          MVI   Buffer,C' '             Clear a buffer to blanks
          MVC   Buffer+1(132),Buffer    Ripple the first blank
          - - -
Buffer    DS    CL133
```

  - ***Problem***: what if the length of the buffer must be changed?

    You must find all occurrences of the symbol **Buffer** and change 132, 133 (and maybe other numbers)

- Solution: define the buffer length in **one** place

```
BufLen    Equ   133                     Define the buffer length
          MVI   Buffer,C' '             Clear a buffer to blanks
          MVC   Buffer+1(BufLen-1),Buffer    Ripple the first blank
          - - -
Buffer    DS    CL(Buflen)
```

  - Advantage: you need to change only the statement defining **BufLen**, and reassemble
  - Instructions don't need to know anything about data declarations

Example 2: Counting characters

- A common instruction sequence:

```
        MVC   Buffer(74),=CL74'A message of about 74 (?) characters...'
```

  - ***Problem***: Some poor soul (you?) had to count the characters to get “74”
    - — Or, didn't want to count, and decided 74 was more than long enough
  - ***Problem***: The MVC instruction must know the length of the literal

- Better: define a constant containing the message:

```
        MVC   Buffer(L'Msg3),Msg3
        - - -
Msg3    DC    C'A message of (I don''t care how many) characters...'
```

- Advantages:
  - The assembler counts the number of characters (correctly!)
  - You can change the message without ever counting again
  - You can add a comments field explaining how and why the message is used (with a literal, you can't)
  - You have more control over where it is placed
  - Instructions don't need to know anything about data declarations

Example 3: Counting characters: using a macro instead

- What you would like:

```
MVC   Buffer(74),=CL74'A message of exactly 74 chars' ←  Old way
MVC2  Buffer,=C'A message of any number of chars'     ←  New way
```

- A simple solution: Define a **MVC2** macro that automatically uses the length attribute of the second operand. For example:

```
        Macro
&Lab    MVC2  &Target,&Source   Prototype statement
&Lab    CLC   0(0,0),&Source    X'D500 0000',S(&Source)
        Org   *-6               Back up to first byte of instruction
        LA    0,&Target.(0)     X'4100',S(&Target),S(&Source)
        Org   *-4               Back up to first byte of instruction
        DC    AL1(X'D2',L'&Source-1)  First 2 bytes of instruction
        Org   *+4               Step to next instruction
        MEnd
```

  - The CLC instruction puts the source operand (possibly a literal) into the assembler's symbol table, so it's available for the L' reference

- The generated instruction is

```
MVC2  Buffer,=C'Message of any...'   HLASM takes care of everything
* * *
MVC   Target(L'&Source),&Source      Just what you wanted!
```

Example 4: Symbols with offsets

- A typical instruction sequence to add inserts in a message or report:

```
          MVC   Buffer+64(12),Insert1   Insert something somewhere
          MVC   Buffer+82(10),Insert2   Insert something somewhere
          - - -
Buffer    DS    CL133
Insert1   DS    CL12                    Inserted data
Insert2   DS    CL10                    Inserted data
```

  - ***Problem***s: if the report must be reformatted, you have to look for *every* offset and length; and, instructions include information about data structures

- Somewhat better: define the insertion points where the **Buffer** is defined

```
Buffer    DS    CL133
          Org   Buffer+64               First insertion point
BufIns1   DS    CL12
          Org   Buffer+82               Second insertion point
BufIns2   DS    CL10
          Org   ,                       Adjust Location Counter
          - - -
          MVC   BufIns1,Insert1         Insert something in a message
          MVC   BufIns2,Insert2         Insert something in a message
```

  - Advantage: no explicit lengths or offsets in the MVC instructions
  - Instructions don't need to know anything about data declarations

Example 4: Symbols with offsets ...

- Much easier: define a DSECT to map the buffer area

```
         USING BuffMap,Buffer    Dependent USING statement
         MVC   BufIns1,Insert1   Insert something in a message
         MVC   BufIns2,Insert2   Insert something in a message
         — — —
Buffer   DS    CL(BuffMapL)
Insert1  DS    CL(L'BufIns1)
Insert2  DS    CL(L'BufIns2)
         — — —
BuffMap  DSECT ,
         DS    CL64              Offset to first insertion point
BufIns1  DS    CL12              First insertion field
         DS    CL6               Position at second insertion point
BufIns2  DS    CL10              Second insertion field
BuffMapL Equ   *-BuffMap         Length of Buffer-mapping DSECT
```

- Advantages:
  - USING instruction overlays **BuffMap** DSECT on **Buffer**
  - No explicit lengths or offsets appear in the MVC instructions
  - Definitions localized to the DSECT
  - Instructions don't need to know anything about data declarations

Example 5: Counting bytes to determine displacements

- An instruction sequence generated by a program-start macro:

```
         Macro
&Name    BEGIN ...various parameters...
         – – –
&Name    Start
         B     102(0,15)      ←— Someone counted exactly 102 bytes!
         DC    17F'0'                          4+17*4=72
         DC    CL20'Assembled &SysDatC. '          +20=92
         DC    CL10'Time &SysTime.'                +10=102
         STM   14,12,12(13)                        All that, just to get here
```

  - ***Problem***: if any change is made, someone has to recount the bytes

- Solution: use the &SYSNDX system variable symbol

```
&Name    Start
         J     S&SysNdx                    The Assembler knows where to go:
         DC    17F'0'
         DC    C'Assembled &SysDatC. '
         DC    C'Time &SysTime. '
         DC    C'At Site &ThisLoc.'        ... Any additional signature
         DC    C'With HLASM &SysVer. '     ... information you want,
         DC    C'on System &System_ID.'    ... without counting bytes
S&SysNdx STM   14,12,12(13)
```

Example 6: Determining record and structure lengths

- Two statement sequences to define a record and its fields:

```
<—— Poor ——————>  <—— Better ——————>
ARecord  DS    0CL923  923?  |    ARecord  DS   0CL(RecLen)
RecHead  DC    H'923'  923?  |    RecHead  DS   Y(RecLen)
Field1   DS    CL44          |    Field1   DS   CL44
Field2   DS    CL55          |             – – –
         – – –               |    RecLen   Equ  *–ARecord
Field999 DS    ...           | ** ASMA080E Statement is unresolvable [?]
```

  - ***Problems***:
    Old: Someone counted the **Field** lengths to determine length "923" (risky!)
    New: HLASM complains about the apparently better symbolic definition
  - **Best:** let the Assembler assign the **Reclen** value for you

```
RecHead  DC    Y(RecLen)     Record length value (as usual)
Field1   DS    CL44
Field2   DS    CL55
         – – –
Field999 DS    CL66
RecLen   Equ   *–RecHead     Define the length
         Org   RecHead       Re–position at start of record
ARecord  DS    0CL(RecLen)   Define name and length of entire record
         Org   ,             Re–position after the record
```

Example 7: Duplicated record definitions

- Code may contain two declarations of the same record structure (say, **OldRec** and **NewRec**)

```
Define Old Record (for Input)                 Define New Record (for Output)
-------------------------------------------   -------------------------------
OldRec   DS    0D                             NewRec   DS    0D
OldType  DS    CL10   Record type             NewType  DS    CL10
OldID    DS    CL4    Record ID               NewID    DS    CL4
OldName  DS    CL40   Name                    NewName  DS    CL40
OldAddr  DS    CL66   Address                 NewAddr  DS    CL66
OldPhone DS    CL12   Phone number            NewPhone DS    CL12
         - - -        etc.                              - - -
         - - -        etc.                              - - -
OldYear  DS    F      Processing year         NewYear  DS    F
OldDay   DS    F      Day of year             NewDay   DS    F
         - - -        etc.                              - - -

         CLC   NewDay,OldDay      Compare record Days (we hope!)
```

- Everything is addressed by current base register(s)
- ***Problem***: Big, BIG trouble if the declarations get out of sync

Example 7: Duplicated record definitions: a better way

- Better: define a *single* DSECT describing the record

```
Record   DSECT ,                  Record description
RecType  DS    CL10               Record type
RecID    DS    CL4                Record ID
RecName  DS    CL40               Name
RecAddr  DS    CL66               Address
RecPhone DS    CL12               Phone number
         - - -                    etc.
RecYear  DS    F                  Processing year
RecDay   DS    F                  Processing day of year
         - - -                    etc.
RecLen   Equ   *-Record           Record length
         - - -
NewRec   DS    0D,CL(RecLen)      Now, define an area for New record
OldRec   DS    0D,CL(RecLen)      Now, define an area for Old record
```

- Advantage: everyone can use the same record definition
  - It can be in a COPY segment, or be generated by a macro
- Next two slides show good techniques to utilize the **Record** DSECT

Example 8: Referencing the records with enhanced USING statements (1)

- Solution 1: With separate base registers for code, and one register for each record instance:
  - Declare Labeled USING statements with qualifiers **Old** and **New**

```
MyProg   CSECT ,                        Resume program control section
         - - -
         LAY   7,OldRec                 Base register for OldRec
         LAY   4,NewRec                 Base register for NewRec
Old      USING Record,7                 Map the Record structure on OldRec
New      USING Record,4                 Map the Record structure on NewRec
         CLC   New.RecID,Old.RecID      Compare record IDs
         JNE   NotThisOne               Go do something else
         MVC   New.RecName,Old.RecName  Copy name field from Old to New
         - - -
```

- A valid complaint: I need two more base registers!
  - Easily fixed, as the next slide shows

Example 8: Referencing the records with enhanced USING statements (2)

- Solution 2: With *existing* base registers for code *and* each record instance (no extra registers!)
  - Declare Labeled Dependent USING statements; the qualifiers are again **Old** and **New**
    - The second USING operand is *relocatable*, not a register number

```
Old      Using Record,OldRec      Map OldRec (labeled dependent USING)
New      Using Record,NewRec      Map NewRec (labeled dependent USING)
         - - -
         MVC   New.RecName,Old.RecName Copy name field from Old to New
```

- The **Record** DSECT is based (or "anchored") on each record field
- Program base register(s) address everything
- This version uses exactly the same base registers as the original (on slide 13), with *no* extra registers needed!
  - And, the program is much easier to update

Example 9: Unreferenced code and data

- Stuff tends to accumulate when no longer referenced or executed
  - ***Problem***: the next person may not be sure something is not needed, so leaves it untouched
  - ***Worse Problem***: a statement label in “dead” code can be an inviting branch target
- ***Solution:*** specify Assembler option **`XREF(SHORT,UNREFS)`** (the default)

```
                Unreferenced Symbols Defined in CSECTs

          Defn  Symbol
            ...
            674 ADDCOM              <— Each unreferenced symbol and the
            724 ADDDIM                 statement where it's defined
           1011 AUTORT
            860 BLANKS
            630 CKDIM
           1038 CLOSE11
```

- If you don't want to delete the statements, skip them:

```
            AGO   .Skip04       Skip the leftovers
            — — —               Unreferenced odds and ends
 .Skip04    ANOP  ,             Intervening statements not assembled
```

- ***Don't*** hide unused statements following the END statement!

Example 10: Register equates and "names"

- Many programs contain EQU statements to "name" registers

```
R0      Equ   0
        — — —
R15     Equ   15
```

  - In the (mistaken) belief that doing so helps you find all register references in the Symbol XREF. Unfortunately, this is NOT true! For example:

```
LM    R14,R12,0(R13)      Refers to all 16 general registers!
```

  ... but only the symbols **R12**, **R13**, and **R14** will appear in the XREF

- Much better: rely on the **Register** XREF (specify the **RXREF** option)
- Another problem: beginners might think register "names" are reserved (as on Intel processors), and write

```
L     R5,R8               Load Register 8 into Register 5 (??)
```

- Usually better just to use register numbers
  - But if your code must refer to multiple register types (general, floating-point, etc.): names *can* help clarify which is used (but *not* with implicit references)

Guidelines for Part 1

- Never count anything yourself; let the Assembler do the work
- Use length attributes rather than hand-counted explicit lengths in SS-type intructions
- Don't write instructions with offsets from symbols; use DSECTs
- Use only a single definition of a record or data structure
  - Let enhanced USING statements clarify references to multiple instances
- Eliminate (or conditionally skip) dead code, unused data and constants
- Keep info about data declarations strictly separate from instructions referencing the data
- Remember: **`EQU`** only gives a name to a value

Part 2: Improving program structure and maintainability

- Ways to cope with ever-expanding programs

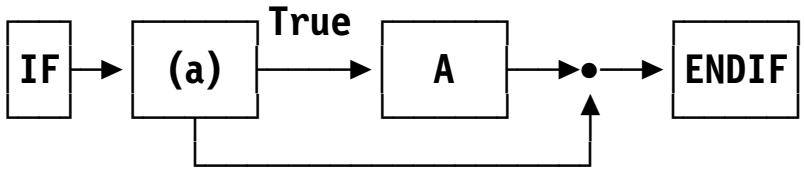
The HLASM Toolkit Structured Programming Macros

- Use the macros to provide uniformity, standardization, maintainability
- Powerful tools to improve program structure, simplify coding
  - Eliminate GOTO statements, extraneous labels, out-of-line logic paths
    - Statement labels represent “unstructured” exposures; each label is a tempting branch target
  - Reduce the number of different constructs used in a program
  - Much easier to understand program flow; no tedious inspection
    - Far less “spaghetti code”
- No more effort to use the SP macros than in a (structured) HLL with GOTOs
- The macros support standard structured-programming forms:

  If-Then-Else, Do, Do-While, Do-Until, Case, Select, Search

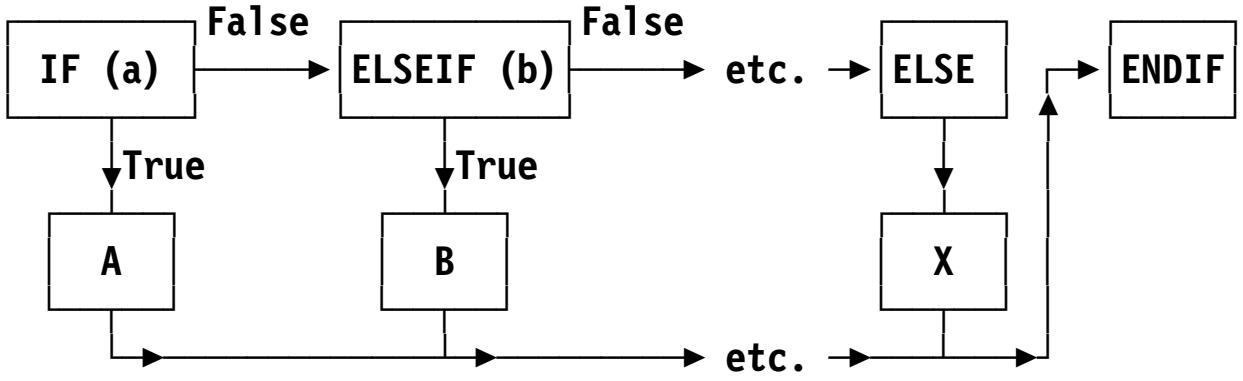
  - All may be fully nested, with multi-level exits
- Some users report SP macros reduce maintenance costs by over 50%

- Example: the basic If-EndIf:




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

- More generally:




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

- Example on the next slide

Simple Example of ordinary vs. structured assembler

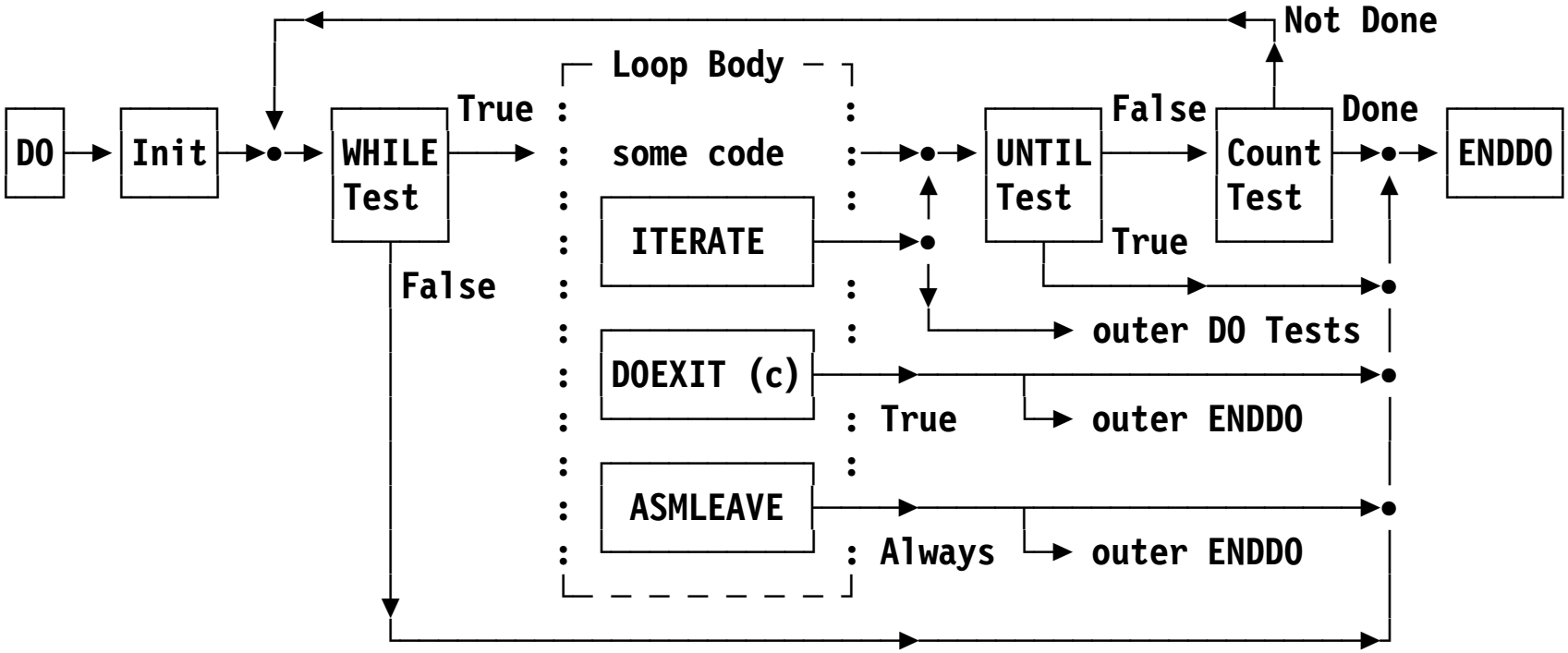
- Ordinary Assembler
  - Lots of labels, branches
  - Typical “spaghetti”

```
         CLC   0(3,2),=C'AAA'
         BNE   T020
         OI    TYPE,TYPEA
         B     T070
T020     CLC   0(5,2),=C'BBBBB'
         BNE   T030
         OI    TYPE,TYPEB
         B     T090
T030     CLC   0(4,2),=C'CCCC'
         BNE   T120
         OI    TYPE,TYPEC
         MVI   SCAN,CFLAG
         B     T150
T070     XC    AWORK,AWORK
         MVI   SCAN,NOTYPE
         B     T150
T090     MVC   BWORK(2),=C'T2'
         B     T150
T120     MVI   TYPE,0
         MVI   SCAN,NOTYPE
T150     DC    0H
```

- Structured Assembler
  - No labels or branches
  - Clearer, simpler

```
    IF     (CLC,=C'AAA',EQ,0(2))
        OI   TYPE,TYPEA
        XC   AWORK,AWORK
        MVI  SCAN,NOTYPE
    ELSEIF (CLC,=C'BBBBB',EQ,0(2))
        OI   TYPE,TYPEB
        MVC  BWORK(2),=C'T2'
    ELSEIF (CLC,=C'CCCC',EQ,0(2))
        OI   TYPE,TYPEC
        MVI  SCAN,CFLAG
    ELSE
        MVI  TYPE,0
        MVI  SCAN,NOTYPE
    ENDIF
```

- Illustrative SP macro: the DO statement supports many combinations of operands



- You can specify very detailed loop controls
- **ITERATE**, **DOEXIT**, and **ASMLEAVE** can specify the label of the current or an outer DO

The HLASM Toolkit Structured Programming Macros ...

- Details are in the *HLASM Toolkit Feature User's Guide*, GC26-8710
  - Macros can generate based or relative branches
- Converting unstructured code
  - You can mix structured and unstructured code
    - Easy to start with **`If-ElseIf-Else-EndIf, Do-EndDo, Case-EndCase`**
    - Start small, and work from the “inside out”
  - Add structure incrementally, leave old code alone if it's too much bother
    - Small changes are quick and easy; programs gradually gain structure
    - “Spaghetti code” is harder to restructure
  - Use constructs that will make it easy to add new cases in the future
  - Major revisions or new programs represent structuring opportunities
  - You can get rid of almost all statement labels: a good thing!
- Remember: conversion is never required!

The incredibly useful and powerful LOCTR assembler instruction

- LOCTR keeps groups of related statements together in the *source*
  - But they need *not* be together in the object code!

```
MyProg   CSect ,                 Control section owning everything
            a...b...c               Statements starting at MyProg
Code     LOCTR ,                 Declare a LOCTR group for instructions
            d...e...f               Some instructions
Data     LOCTR ,                 Declare a LOCTR group for data
            p...q...r               Data, constants, etc.
Literals LOCTR ,                 Declare a LOCTR group for literals
Code     LOCTR ,                 Resume the CODE LOCTR group
            g...h...j               More instructions
Data     LOCTR ,                 Resume the DATA LOCTR group
            s...t...u               More data, constants
Literals LOCTR ,                 Declare a LOCTR group for literals
         LTORG ,                    Your literals
```

  - HLASM sorts the groups in the order they're declared:

```
MyProg
            a...b...c
Code        All items in the 'Code' LOCTR group
            d...e...f...g...h...j
Data        All items in the 'Data' LOCTR group
            p...q...r...s...t...u
Literals    All items in the 'Literals' LOCTR group
```

Minimizing base register requirements

```
Program CSect ,
Entry   J     Start
-------------------------------
Consts  LOCTR ,
     - - - Constants - - -
-------------------------------
Lits    LOCTR ,
-------------------------------
Work    LOCTR ,
     - - - Work Area - - -
     (if not reenterable)
-------------------------------
Code    LOCTR ,
Start   Save  (14,12)  Etc.
        LR    12,15
        Using Entry,12
        - - -
*   remainder of program,
*   using relative branch
*   instructions and NO
*   code-base registers
-------------------------------
Lits    LOCTR ,
        LTORG ,
     - - - Literals  - - -
```

1. Use LOCTR to cause data items to be grouped together at the start of the CSECT
2. Use only relative branches among instructions in the program area
   - Use EXRL for EXecute instructions
3. If necessary, use LAY and LARL to reference constants, literals, and work area items

There's no need for “code base” registers; base register(s) are needed *only* for constants, literals, and the work area!

Using the LOCTR instruction: example

- Example of source statements:

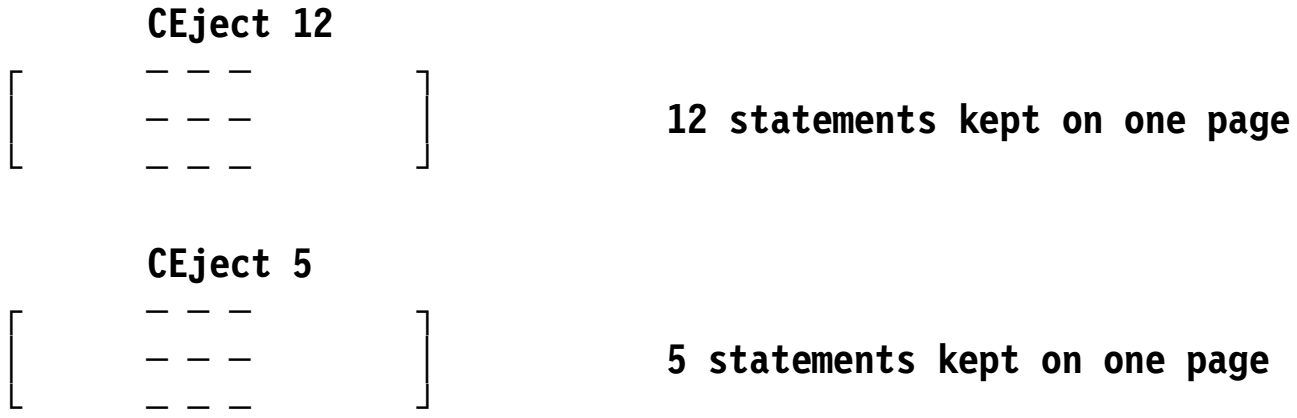
```
Loc          Statement
0000 ..  1 LocXmp CSect ,
0000 ..  2        J     Start      To Code area
0004 ..  3 Const  Loctr ,          Constants
0008 ..  4 Lits   Loctr ,          Literals
000C ..  5 Work   Loctr ,          Work Area
000D ..  6 Code   Loctr ,          Program Code
     ..  7 Start  Save  (14,12)    Etc.
000E ..  8+Start  DS    0H
000E ..  9+       STM   14,12,12(13)
0012 .. 10        LR    12,15      Copy base reg
     .. 11        Using LocXmp,12
0014 .. 12        L     15,=V(Sub)
0018 .. 13        BASR  14,15
001A .. 14        C     15,RetVal
0004 .. 15 Const  Loctr ,
0004 .. 16 RetVal DC    A(47)      Return code
001E .. 17 Code   Loctr ,
001E .. 18        JE    OKRet      Jump if OK
0022 .. 19        MVI   Flag,X'FF'
000C .. 20 Work   Loctr ,
000C .. 21 Flag   DS    X
0026 .. 22 Code   Loctr ,
0026 .. 23 OkRet  LA    2,Flag
     .. 24 *      - - -            More code
0008 .. 25 Lits   Loctr ,
0008 .. 26        LtOrg ,
0008 .. 27                =V(Sub)
     .. 28 *      - - -            More anything
     .. 29        End
```

- How HLASM arranges the *object* code:

```
Loc          Statement
0000 ..  1 LocXmp CSect ,
0000 ..  2        J     Start
     - - -  Consts first - - -
0004 .. 16 RetVal DC    A(47)
     - - -  Lits next - - -
0008 .. 27                =V(Sub)
     - - -  Work next - - -
000C .. 21 Flag   DS    X
     - - -  Code last - - -
     ..  7 Start  Save  (14,12)
000E ..  8+Start  DS    0H
000E ..  9+       STM   14,12,12(13)
0012 .. 10        LR    12,15
     .. 11        Using LocXmp,12
0014 .. 12        L     15,=V(Sub)
0018 .. 13        BASR  14,15
001A .. 14        C     15,RetVal
001E .. 18        JE    OKRet
0022 .. 19        MVI   Flag,X'FF'
0026 .. 23 OkRet  LA    2,Flag
     .. 24 *      - - -
     .. 28 *      - - -
     .. 29        End
```

CEJECT for improved listing readability

- Listings don't always keep related groups of statements together on the same page
  - Automatic page ejects can make program listings harder to read
- Use CEJECT (“Conditional EJECT”) to keep them grouped



  - CEJECT counts lines remaining on the page, EJECTs if not enough
- Improved readability improves understanding!

Advice from experienced (and very successful!) programmers

- A consistent overall style of program organization is valuable
  - Use comments generously
  - Naming conventions should make it easy to identify modules, files, records, fields, statement labels, macros, subroutines, etc.
  - Use subroutines frequently
    - With consistent conventions for linkage, argument passing, and addressability
    - Any subroutine should be able to call any other
  - Keep routines to manageable size (1-3 pages max)
- Important guidelines:
  - *Don't* use EQU for statement-label creation
  - *Do* use extended mnemonics (except when there isn't one; then, use meaningful EQUated symbols for the mask values)
  - *Never* use label offsets (like **X+6** or ***+8**), especially for branches
  - *Don't* write explicit lengths when they're the same as length attributes
- Anything that degrades program understandability is bad
- Gains in simplicity greatly outweigh any apparent performance cost

Part 3: Benefiting from newer, efficient instructions

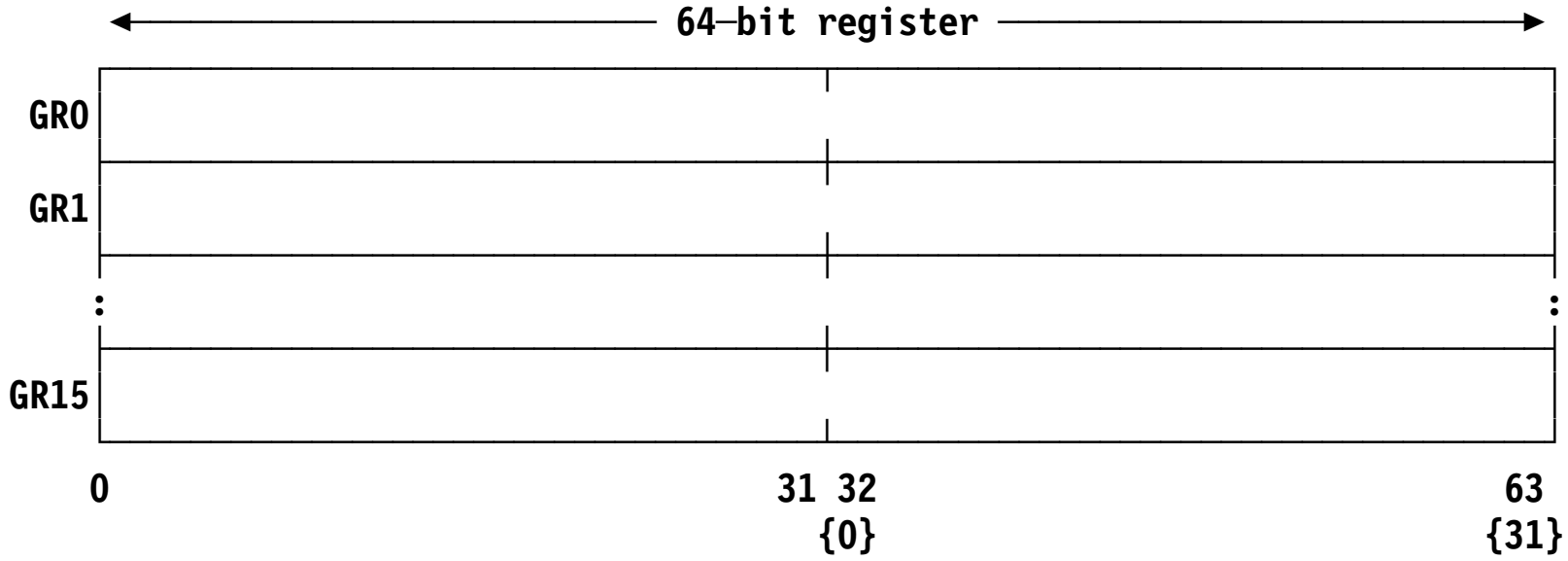
- Many new easy-to-use instructions
  - Minimize the costs of memory references
  - Fewer base registers are needed
  - Statements are easier to understand

Topics

- Quick review of some z/Architecture features
- New forms of address generation -- reduce the number of base registers
  - Base and unsigned 12-bit displacement
  - Base and signed 20-bit displacement
  - Instruction-relative addressing
- Instructions with immediate operands -- fewer memory references
  - Load and insert instructions
  - Arithmetic instructions
  - Logical instructions
- Some instructions worth knowing about

Reminder: 64-bit general registers

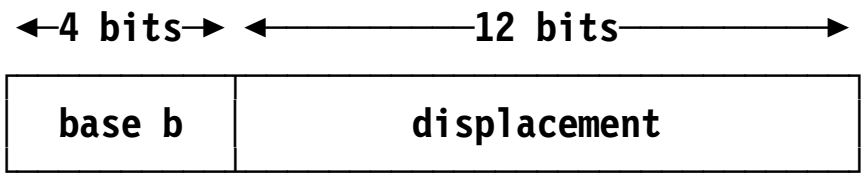
- z/Architecture general registers are 64 bits long:



- Old instructions use only the right halves (left halves are “invisible”)
- Rightmost bits are numbered either 32-63 (or sometimes {0-31})
- You can use 64-bit registers ***without*** needing 64-bit addressing mode!

Review of base-displacement address generation

1. With unsigned 12-bit displacement



   - Effective Address = displacement + [***if*** (b ≠ 0) ***then*** c(GRb)]
   - Provides addressability to at most 4096 bytes per base register
     - And, you can't address anything preceding the generated address

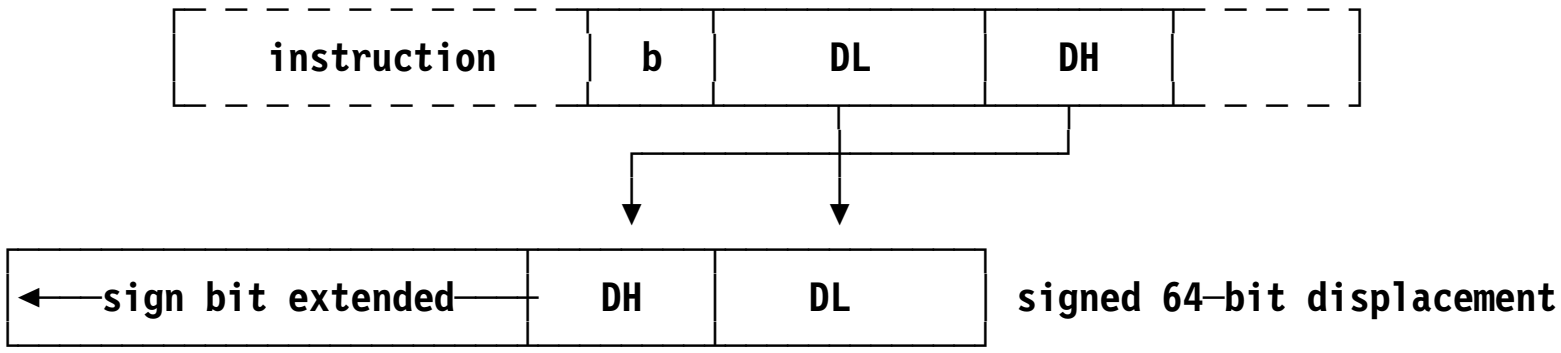
2. With signed 20-bit displacement
   - New instruction format:

opcode	R_1	X_2	B_2	DL_2	DH_2	opcode

   - Traditional unsigned 12-bit displacement field now named **DL_2**
   - High-order 8-bit signed displacement extension named **DH_2**

Address generation with base and signed 20-bit displacement

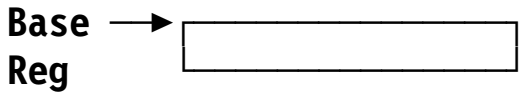
- 20-bit signed displacement formed from DH and DL:
  - DH concatenated at high end of DL and then sign-extended to 64 bits



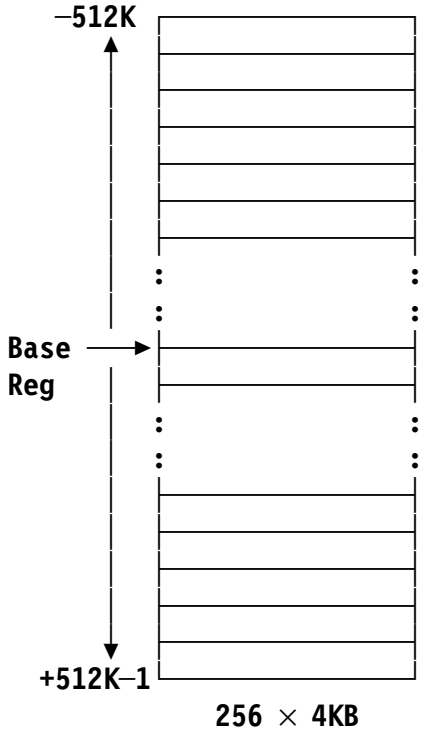
  - Displacement range $(-2^{19}, +2^{19}-1)$ rather than (0,4095)
- Address calculation adds base/index register contents as appropriate
  - Number of significant address digits depends on current addressing mode
- If the DH field is zero, get usual 12-bit displacement

Benefits of 20-bit displacements

- Very large data structures addressable with a single base register
  - Addresses 1MB (±512KB) per base register
- Base register can now point to the middle of a data structure
- 12-bit displacement addresses only 4KB



  - Addressing 1MB could require 256 base registers...



- Fewer base registers are needed to address large areas!

Review of HLASM USING base-displacement resolution rules

1. Expression and USING-table entry relocatability attributes must match
2. Calculate possible displacements; choose smallest non-negative
3. If no non-negative displacements are available, use smallest negative value
4. If more than one such smallest displacement, choose higher-numbered register

```
000000                  00000 00012    1 Test CSECT ,
                  R:AB  00000          2      Using *,10,11
                        00000          3 X    Equ   *
000000 E300 B880 1208         13880    4      AG    0,X+80000  Long displacement
000006 E300 AFA0 0008         00FA0    5      AG    0,X+4000   R11 +96 bytes away
                                       6      Drop  10
00000C E300 BFA0 FF08         00FA0    7      AG    0,X+4000   Negative displacement

                                       8 *    Note absolute displacements:
000012 E300 0120 7A71          7A120    9      LAY   0,+500000
000018 E300 0EE0 8371         F83000   10      LAY   0,-512000  AMode sensitive!
```

- DH fields in the object code are underscored

Relative addressing

- New instruction formats with 2-byte and 4-byte immediate operands, 12-bit opcodes
- 4-byte instruction:

opcode	R_1	op	RI_2

RI_2 range: $-2^{15} \leq I_2 \leq 2^{15}-1$, or $-32768 \leq I_2 \leq 32767$

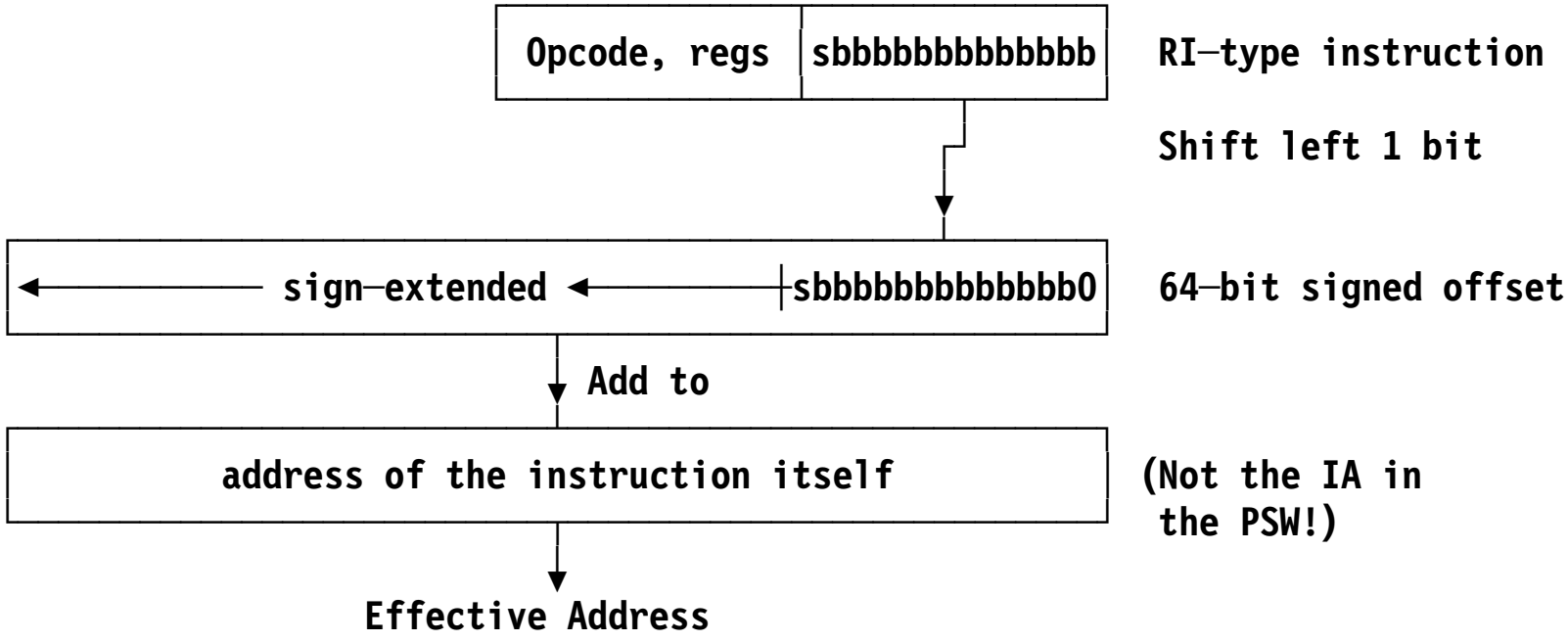
- 6-byte instruction:

opcode	R_1	op	RI_2

RI_2 range: $-2^{31} \leq RI_2 \leq 2^{31}-1$, or $-2147483648 \leq RI_2 \leq 2147483647$

Relative addressing ...

- Address generation:



- RI_2 operand is *doubled* because a branch target is always on an even boundary
- **No** base register is used; base register requirement(s) can be minimized

Important relative branch instructions

- Branch Relative on Condition:

A7	M_1	4	RI_2

The branch target can be as far as −65536 and +65534 bytes away (±64K)

- Branch Relative Long on Condition:

C0	M_1	4	RI_2

The distance to the branch target can be up to 4 billion bytes from the RIL-type instruction, in either direction. (±4G ... enough for now?)

Operation	**Immediate-Operand Length**	
	16 bits	**32 bits**
Branch on Condition (Relative)	`BCR [JC]`	`BCRL [JLC]`

- Extended mnemonics in [square brackets] start with `J` (for “Jump”)

Relative branch on condition instructions and extended mnemonics

RI Mnemonics		RIL Mnemonics		Mask	Meaning
BRC	JC	BRCL	JLC	M_1	Conditional Branch
BRU	J	BRUL	JLU	15	Unconditional Branch
BRNO	JO	BRNOL	JLNO	14	Branch if Not Ones (T) Branch if No Overflow (A)
BRNH	JNH	BRNHL	JLNH	13	Branch if Not High (C)
BRNP	JNP	BRNPL	JLNP	13	Branch if Not Plus (A)
BRNL	JNL	BRNLL	JLNL	11	Branch if Not Low (C)
BRNM	JNM	BRNML	JLNM	11	Branch if Not Minus (A) Branch if Not Mixed (T)
BRE	JE	BREL	JLE	8	Branch if Equal (C)
BRZ	JZ	BRZL	JLZ	8	Branch if Zero(s) (A,T)
BRNZ	JNZ	BRNZL	JLNZ	7	Branch if Not Equal (C)
BRNE	JNE	BRNEL	JLNE	7	Branch if Not Zero (A,T)
BRL	JL	BRLL	JLL	4	Branch if Low (C)
BRM	JM	BRML	JLM	4	Branch if Minus (A) Branch if Mixed (T)
BRH	JH	BRHL	JLH	2	Branch if High (C)
BRP	JP	BRPL	JLP	2	Branch if Plus (A)
BRO	JO	BROL	JLO	1	Branch if Ones (T) Branch if Overflow (A)
	JNOP		JLNOP	0	No Operation

- (A) = after arithmetic, (C) = after comparison, (T) = after test
  - Be careful: JLxx means “Jump Long“, not “Low”

“Branch Relative and Save” instructions

- No base registers or address constants needed!

Operation	Immediate-Operand Length	
	16 bits	32 bits
Branch and Save (Relative)	BRAS [JAS]	BCRL [JASL]

- Example of a local subroutine:

```
JAS   12,LocalSub        Link to internal subroutine
```

- Operands can be *external* references! For example:

```
EXTRN BigSub
JAS   12,BigSub          (Small load module or program object)
```

or

```
JASL  12,BigSub          (Large load module or program object)
```

- No address constants required; z/OS Binder resolves the relative offsets

Other useful relative-address instructions

EXRL Execute Relative Long: no base register required

LARL Load Address Relative Long: no base register required (but the target must be at an even address)

- Loop control instructions:

Operation	Register Length	
	32 bits	**64 bits**
Branch on Count (Register)	BCTR	BCTGR
Branch on Count (Indexed)	BCT	BCTG
Branch on Count (Relative)	BRCT [JCT]	BRCTG [JCTG]
Branch on Index	BXH BXLE	BXHG BXLEG
Branch on Index (Relative)	BRXH [JXH] BRXLE [JXLE]	BRXHG [JXHG] BRXLG [JXLEG]

Compare and branch instructions

- Compare and branch instructions combine the two operations:

CRB, CGRB	Compare and Branch	CRJ, CGRJ	Compare and Branch Relative
CIB, CGIB	Compare Immediate and Branch	CIJ, CGIJ	Compare Immediate and Branch Relative

  - The I_2 (comparand) operand is a signed 8-bit number
  - All instructions support extended mnemonics

```
Old Ways                          Better Ways

CR    3,4
JNE   NotSame                     CRJNE 3,4,NotSame

C     9,=F'-99'
JL    TooSmall                    CIJL  9,-99,TooSmall

LTR   0,0
JNM   NotMinus                    CIJNM 0,0,NotMinus
```

  - CRB, CGRB, CIB, and CGIB are based branches
- Save (several) instructions and memory references

What about macros that generate base-displacement instructions?

- Relative branches may have eliminated the need for base registers for your code, but...
- Many IBM macros generate based instructions like BAL, BC, LA, ST
  - ***Solution 1***: Issue the **SYSSTATE** macro

```
         SYSSTATE ARCHLVL=1       Enables immediate and relative ops
         SYSSTATE ARCHLVL=2       Enables z/Architecture ops
```

  - ***Solution 2***: Create a temporary local base register:

```
         PUSH  USING              Save current USING status
         BASR  tempreg,0          Any unused register in (2,12)
         USING *,tempreg          Temporary local addressability
        <Macro Invocation>        Expand the macro
         POP   USING              Restores previous USING status,
*                                 DROPs 'tempreg' automatically
```

- Some self-modifying macro expansions can be placed in the same area as constants and work areas
  - Or, use MF=L form for skeletons, and inline MF=E forms for execution

Load Address instructions LAY and LARL

- Addressing local items without a memory reference to a literal
- The old (slow) way:

```
L     1,=A(Target)
```

- New, faster way with LAY and LARL:

```
LARL  1,Target          If Target is known to be at an even address
```

or

```
LAY   1,Target          If Target might be at an odd address
```

or

```
BASR  anyreg,0          Set a temporary base address
USING *,anyreg          Temporary addressability
LAY   1,Target          Target known to be at odd address
DROP  anyreg
```

- Suppose you need to set the high-order bit of a linkage address:

Old Way

```
L     15,=A(Sub)
O     15,=X'80000000'
BASSM 14,15
```

Better Way

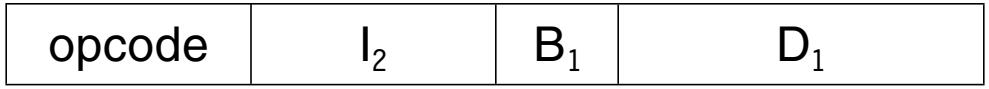
```
LARL  15,Sub
OILH  15,X'8000'
BASSM 14,15
```

  - LARL also is used for external routines if **Sub** is declared in an **EXTRN** statement

Immediate Operands

- We're familiar with SI-type instructions with “immediate” operands:

.



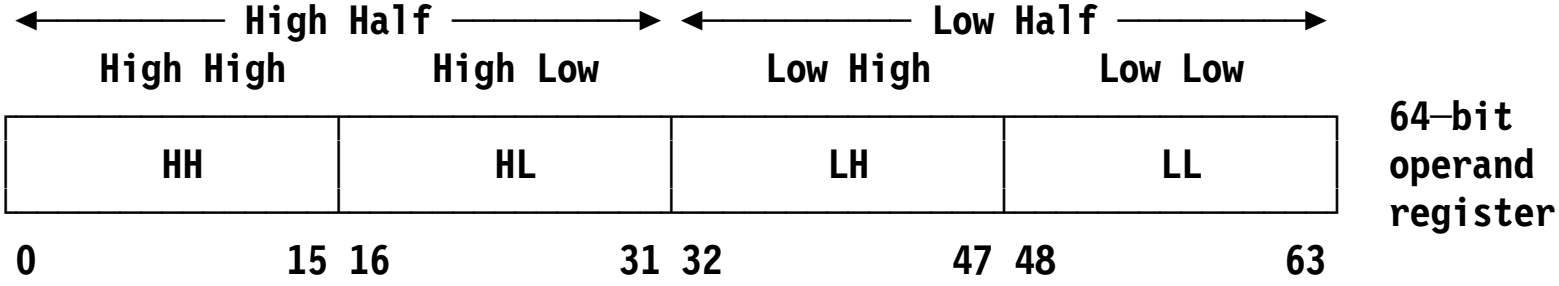
  - Used for instructions with logical operands, like MVI, CLI, TM, OI, etc.
- Many new instructions with 16- or 32-bit immediate operands:

.

  - Signed *and* unsigned arithmetic
  - Logical operations (up to 32 bits)
  - Branch relative (no base register required!)
- Greater flexibility, many different operand types
- Help you save memory, reduce memory references, free up registers

Register segments

- Some instructions refer to 16- or 32-bit portions of the 64-bit register:



- Last one or two letters of many instruction mnemonics indicate which part of the register is involved:

HH **H**igh Half's **H**igh Half (bits 0-15)
HL **H**igh Half's **L**ow Half (bits 16-31)
LH **L**ow Half's **H**igh Half (bits 32-47)
LL **L**ow Half's **L**ow Half (bits 48-63)

H **H**igh Half (bits 0-31)
L **L**ow Half (bits 32-63)

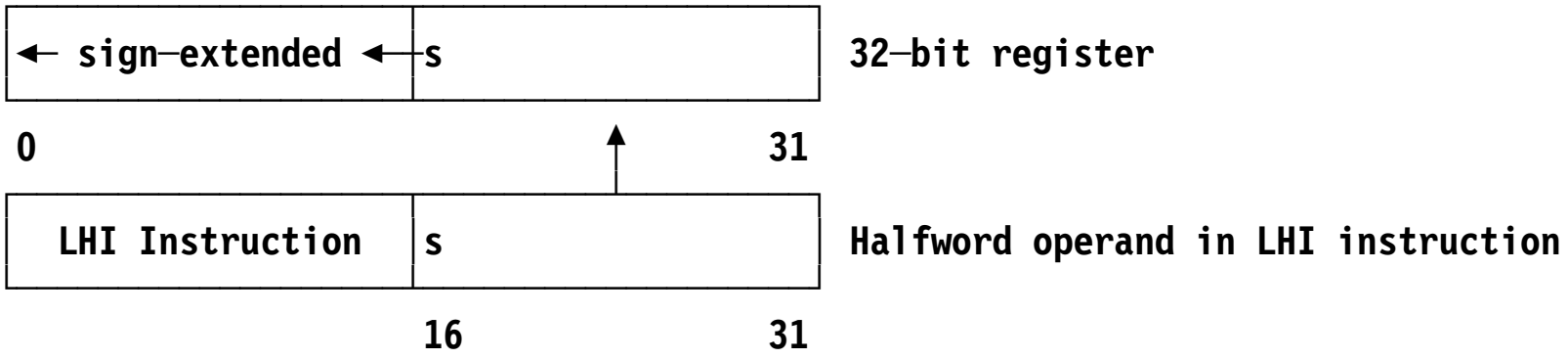
Load and insert instructions with immediate operands

- Arithmetic load instructions extend the immediate-operand sign
- Logical load instructions don't extend; set the rest of the register to zero
- Insert-immediate instructions don't affect any part of the target register other than bit positions where the immediate operand was inserted.

Operation	Operand 1	32-bit register		64-bit register	
	Operand 2	16 bits	32 bits	16 bits	32 bits
Arithmetic Load		LHI		LGHI	LGFI
Logical Load				LLIHH LLIHL LLILH LLILL	LLIHF LLILF
Insert		IILH IILL	IILF	IIHH IIHL	IIHF

- Faster and simpler than old instruction sequences

Examples using load-immediate instructions



- Eliminate memory references and constants in storage

```
Old Ways                        Better Ways
L     1,=F'275'                 LHI   1,275
LH    2,=H'-5678'               LHI   2,-5678
L     3,=F'123456789'           LGFI  3,123456789  (64-bit register)
                                IILF  3,123456789  (32-bit register)
```

- Eliminate unnecessary register zeroing, needless memory references

```
Old Way                         Better Way
SR    1,1
ICM   1,B'11',=C'AB'            LLILL 1,C'AB'
```

- Faster operation, smaller programs, no base register needed

Arithmetic instructions with immediate operands

Operation	Operand 1	32-bit register		64-bit register	
	Operand 2	16 bits	32 bits	16 bits	32 bits
Arithmetic Add/Subtract		AHI	AFI	AGHI	AGFI
Logical Add/Subtract			ALFI SLFI		ALGFI SLGFI
Arithmetic Compare		CHI	CFI, CRL	CGHI	CGFI, CGFRL
Logical Compare			CLFI		CLGFI
Multiply		MHI		MGHI	

- Instructions referencing 32-bit registers are immediately useful

Old Ways		Better Ways	
A	6,=A(Offset*4)	AFI	6,Offset*4
CH	4,=H'-1'	CHI	4,-1
MH	2,=Y(ItemLen)	MHI	2,ItemLen
CL	9,=X'107429B3'	CLFI	9,X'107429B3'

- Faster operation, smaller programs, no base register needed

Immediate instructions for logical operations on registers

- Instructions operate on 32 bits of a 64-bit register, or on 16-bit high or low halves of each half

Operation	Operand 1	64-bit register	
	Operand 2	16-bit immediate operand	32-bit immediate operand
AND		NIHH, NIHL NILH, NILL	NIHF NILF
OR		OIHH, OIHL OILH, OILL	OIHF OILF
XOR			XIHF XILF
Test Under Mask		TMHH, TMHL TMLH, TMLL	

- Underscored instructions operate within the rightmost 32 bits
  - Exercise for the reader: why are the AND and OR instructions with 16-bit operands unnecessary?

Immediate instructions for logical operations on registers ...

- Isolate the rightmost 6 bits of GR4:

```
Old Ways                          Better Way
N     4,=X'0000003F'              NILL  4,X'3F'
SLL   4,26
SRL   4,26                        NILL  4,X'3F'
SRDL  4,6  (lose R5 bits!)
SR    4,4
SLDL  4,6                         NILL  4,X'3F'
```

- Can the 31-bit-mode address in R5 refer to items below the 16M line?

```
Old Way                           Better Way
LR    0,5                         TMLH  5,X'7F00'
SLL   0,1                         JZ    Its_Safe
SRA   0,25
JZ    Its_Safe
```

- Is the integer in register 9 a multiple of 4?

```
Old Way                           Better Way
LR    0,9                         TMLL  9,X'0003'
N     0,=A(X'3')                  JZ    Mult4
JZ    Mult4
```

- In each case: extra register(s), extra instruction(s), or memory reference(s)

Conditional load and store instructions

- Load or store action depends on Condition Code setting

Operation	Operand length	
	32 bits	64 bits
Load Register on Condition	LROC	LROCG
Load on Condition	LOC	LOCG
Store on Condition	STOC	STOCG

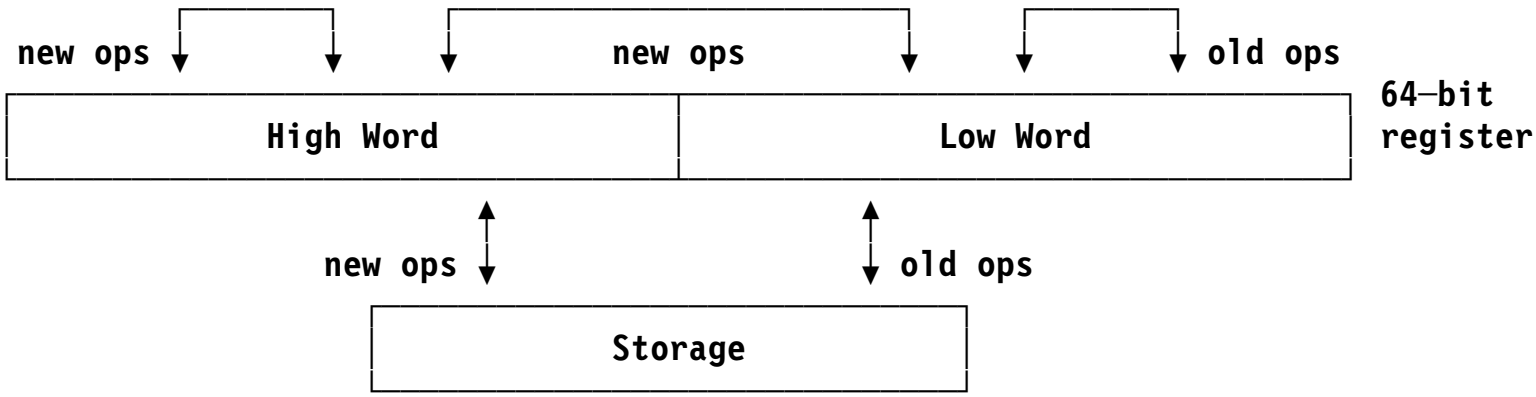
  - All have extended mnemonics: append **E/NE**, **H/NH**, **L/NL**

- Example: put larger value from registers 0 and 1 into register 2

```
     Old Way                                 New Way
     LR   2,0    Guess c(R0)>=c(R1)          LR     2,0    Guess
     CR   0,1    Compare                     CR     0,1    Compare
     JNL  OK     Branch if correct           LROC   2,1    Load if c(R0)<c(R1)
     LR   2,1    No, C(R0)<c(R1)
OK   - - -
```

- Reduces number of branch instructions and flow paths
  - CPU need not do branch prediction or update Branch History Table

High-word instructions: 16 more 32-bit work registers!



- Many low-word operations available for high word
  - Many high ← high, high ← low, and low ← high operations
  - 48 new instructions, plus many extended mnemonics
- Use low words for base registers, addressing; high words for other work
- Example:

```
          L     8,Table_Addr      Base address in low half of R8 (R8L)
          LFH   8,Loop_Count      Iteration count in high half of R8 (R8H)
LoopHead  L     4,0(0,8)          Get some data...
          - - -                   Work on it
          BRCTH 8,LoopHead        Count down in R8H and iterate
```

Distinct-operand instructions

- Many “traditional” instructions overwrite the initial value of the target operand

```
SLL   2,12          Original contents of R2 changed
AR    4,7           Original contents of R4 changed
```

- New “distinct-operand” instructions add “**K**” to the original mnemonic

```
SLLK  0,2,12        Result in R0; contents of R2 unchanged
ARK   3,4,7         Sum in R3; contents of R4 unchanged
```

- These instructions let you preserve a value without first copying it:

```
Old Way             New Way
LR    3,4           ARK   3,4,7
AR    3,7
```

Useful instructions: MVCOS, byte reversal

- You don't need **EX** for variable-length moves!
- **MVCOS**: Move with Optional Specifications
  - Operand format: $D_1(B_1),D_2(B_2),R_3$
  - Both operands use base-displacement addressing (like MVC)
- Non-privileged use: just set GPR 0 to zero!
- Moves up to 4K bytes of data from anywhere to anywhere
  - If all bytes moved, CC=0; otherwise, CC=3
  - If moving more than 4K: update addresses and length, and loop
  - No checks for destructive overlap

- Byte-Reversing Load/Store for handling wrong-endian data

Instructions	Operation
LRVH, STRVH	Transmit 2 rightmost bytes in reverse order
LRV, LRVR, STRV	Transmit 4 rightmost bytes in reverse order
LRVG, LRVGR, STRVG	Transmit 8 rightmost bytes in reverse order

- Rightmost bytes of register used for 2-byte and 4-byte ops
- For Load operations, remaining bytes in register are unchanged

Guidelines for Part 3

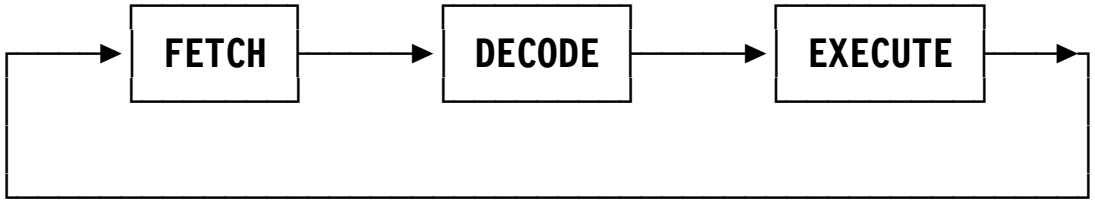
- Memory references are potential “performance poison”
- Use instructions with immediate operands instead of instructions referencing storage operands (such as short literals) wherever possible
- Use relative branch instructions to minimize base-register requirements
- Long-displacement instructions provide very large addressability range
  - Can reduce the need for multiple base registers
- High-word instructions provide an additional 16 “work” registers
- Many new instructions can eliminate or reduce memory accesses and conflicts

Part 4: Enhancing awareness of CPU behavior

- These items may be useful for CPU-intensive or frequently-executed programs

Processor evolution

- Conceptual CPU behavior (the way we learned it):
  1. Fetch the instruction from memory
  2. Decode it and get the operands
  3. Execute the instruction and put away the results



- You can still think of it that way, but...
- Modern CPUs split each of these steps into many overlapped stages in a “pipeline”
  - Anything that affects pipeline flow will slow execution
  - There are many conditions that affect performance at the instruction level

Memory caches

- Memory speed is *very* slow compared to CPU speed
- Instructions and data are therefore “cached” in processor-controlled high-speed buffers, for faster access
  - Cache elements are usually called “lines”; typically 256 bytes
- Cache is to main storage as (virtual) main storage is to paging storage
  - The concepts of “thrashing”, “working set”, and “locality of reference” apply also to the cache
- Operand alignment can be very important!
  - The CPU handles misaligned operands; but if the data spans a doubleword boundary, cache line, or page, the operation can be *much* slower
  - Try not to cross doubleword boundaries if possible
  - Source operands should be on or within a doubleword boundary
- It's important to understand how your code can affect cache behavior

Mixing code and work areas: a poor practice

- Occasionally programs will mix instructions and read/write work areas:

```
          CVD   3,DWork                  Convert to decimal
          UNPK  DWork+4(4),Temp(7)       Unpack to EBCDIC
          OI    Temp+6,X'F0'             Set correct zone on last digit
          J     NextTask                 Go do something useful with it
          - - -
DWork     DS    D                        Work area, mixed with code!
Temp      DS    CL7                      EBCDIC result
NextTask  DC    0H
          MVC   Somewhere(L'Temp),Temp   Move the result
          - - -
```

- Severe impact on performance
  - New systems have separate instruction and data caches
  - CPU must flush and reload the instruction cache if anything is stored into the cache line
    - And maybe the next one, if it has prefetched instructions far enough ahead
  - Unfortunately many standard macro expansions mix code and data
    - Use List and Execute forms if performance is important
- If readability is that desirable, use LOCTR instead

Interlocks

1. Address-generation interlock (AGI): waiting for an operand address

Old Way

```
LA   3,1(,3)   Bump pointer
IC   0,0(,3)   Get a byte (wait!)

L    7,=A(Data)
L    1,0(,7)   Get a value (wait!)
- - -      Other instructions
- - -
```

New Way

```
IC    0,1(,3)
LA    3,1(,3)

L     7,=A(Data)
- - -      Unrelated instructions
- - -
L     1,0(,7)  Get a value
```

   - AGI also affects based branch instructions

2. Instruction-fetch interlock (IFI): don't modify code! CPU must flush the entire instruction cache and pipeline, and start up again

Old Way

```
BC   0,InitDone     Skip initialization
OI   *-3,X'F0'      Make a branch
```

   - Better: set a flag bit in a work area

Interlocks ...

3. Operand store compare: CPU waits for a result to arrive in memory, only to fetch it again

```
Old Way                          New Way
ST   2,Result                    ST   2,Result
CLC  Result,OldValue             CL   2,OldValue

MVC  WorkArea(8),Data            MVC  WorkArea(8),Data
CLI  WorkArea+7,C'A'             CLI  Data+7,C'A'
```

4. Instruction decoding continues (including possible branch paths) ahead of currently executing instruction
   - CPU tries to predict the next instruction path(s)
     - Some instructions are predicted to always branch: **BC 15, BCT/BCTG, BXLE/BXLEG**
   - Always try to arrange branches so the “fall-through” case is most likely

```
LTR   15,15          Check for error
JNZ   Error_27       Branch only on unusual condition
- - -               Continue normal processing
```

Guidelines for Part 4

- Keep data correctly aligned to avoid cache (and page) thrashing
- Address data sequentially rather than randomly
- Don't mix code and read/write data areas
  - Keep them as far apart as you (reasonably) can
- Keep referenced data close in memory *and* in time
  - Keep data frequently read (but infrequently updated) separate from data frequently updated
- Minimize branches
  - Keep serialized objects on separate cache lines
- Keep your code compact, and avoid unnecessary branches
- *Strenuously* avoid modifying instructions, and don't construct them to be executed (or inserted into the instruction stream)
- Keep EXecute targets very close to the EXecuting instruction (EX, EXRL)
- Use long-displacement instructions judiciously
- Start critical loops on a doubleword (or stricter) boundary
- Use QSAM for I/O: it has been highly optimized

Summary

What's good about Assembler Language?
Almost everything you do works OK

What's bad about Assembler Language?
Almost everything you do works OK

Things worth remembering

1. Never count things yourself; let HLASM do the work for you
   - This includes the “length” operands of SS-type instructions
   - If anything changes, you won't have to find the old counts
2. Memory references are increasingly expensive
   - Use instructions with immediate operands wherever possible
3. Closely mixing instructions and data is expensive
   - Modifying instructions can be ***very*** expensive (especially if they are executed repeatedly)
4. Group constants and literals together; do the same for work areas
   - Locality of reference helps performance; LOCTR makes it easy
5. Look for opportunities to use new instructions
6. ***Anything*** that improves understandability is a good thing
7. Don't be too clever!
   - You may not remember why you did it that way three months from now
   - Pity the poor programmer who has to fix it, and figure out what you did

Subscribing to ASSEMBLER and IBM-MAIN Discussion Groups

- These two lists are monitored by experienced, helpful people
- ASSEMBLER-LIST
  - Send e-mail to

    LISTSERV@LISTSERV.UGA.EDU

    with no subject line, and a single-line message saying

    SUBSCRIBE ASSEMBLER-LIST <your name>

- IBM-MAIN
  - Send e-mail to

    LISTSERV@LISTSERV.UA.EDU

    with no subject line, and a single-line message saying

    SUBSCRIBE IBM-MAIN <your name>

- You'll receive a confirmation message with more info.

Useful references

- “How to Benefit From HLASM's Most Powerful Features”. SHARE Aug. 2011, Session 9223
- “Structured Assembler Language Programming Using HLASM: Not Your Father's Assembler Language”. SHARE Aug. 2009, Session 8133
- “Reducing Base Register Utilization: How to ‘Jumpify’ Your Programs”. SHARE Feb. 2011, Session 8548
- “A ‘Quick Start’ Approach to Training Anyone to Write Assembler Language”. SHARE Mar. 2009, Session 8144
- “User Experience – Tuning Old Assembler Code to Exploit z/Architecture”. SHARE Feb. 2007, Session 8185
- “How Do You Do What You Do When You're a CPU? Two.” SHARE Feb. 2005, Session 2835
- IBM documentation:
  - High Level Assembler Language Reference SC26-4940
  - High Level Assembler Programmer's Guide SC26-4941
  - High Level Assembler Toolkit Feature User's Guide GC26-8710
  - z/Architecture Principles of Operation SA22-7832
  - z/Architecture Reference Summary SA22-7871