## Table of Contents

### Part 1: Tidying up your programs
- Counting Characters .................................................. 2
- Counting characters vs. a simple “MVC2” macro .......... 3
- Initializing a buffer .................................................. 4
- Counting bytes to determine displacements .............. 5
- Determining record and structure lengths ................. 6
- Symbols with offsets .................................................. 7
- Duplicated record definitions .................................... 8
- Duplicated record definitions: a better way .......... 9
- Enhanced USING Statements ........................................ 10
- Unreferenced code and data ...................................... 11
- Register equates and “names” ................................. 12

### Part 2: Benefiting from newer, efficient instructions
- Topics .......................................................................... 13
- Immediate Operands .................................................. 14
- Register segments ..................................................... 15
- Load and insert instructions with immediate operands .. 16
- Examples using load-immediate instructions ............ 17
- Arithmetic instructions with immediate operands .. 18
- Immediate instructions for logical operations on registers 19
- Review of base-displacement address generation .......... 20
- Address generation with base and signed 20-bit displacement 21
- Benefits of 20-bit displacements ............................ 22

---

**Upgrading Assembler Programs**

SHARE 117, Session 9281
<table>
<thead>
<tr>
<th>Contents-2 Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Review of HLASM USING base-displacement resolution rules</td>
</tr>
<tr>
<td>Relative addressing</td>
</tr>
<tr>
<td>Important relative branch instructions</td>
</tr>
<tr>
<td>Relative branch on condition instructions and extended mnemonics</td>
</tr>
<tr>
<td>Other useful relative-address instructions</td>
</tr>
<tr>
<td>Compare and branch instructions</td>
</tr>
<tr>
<td>What about macros that generate base-displacement instructions?</td>
</tr>
<tr>
<td>Conditional load and store instructions</td>
</tr>
<tr>
<td>High-word instructions: 16 more 32-bit work registers!</td>
</tr>
<tr>
<td>Distinct-operand instructions</td>
</tr>
<tr>
<td>**Part 3: Enhancing awareness of CPU behavior**</td>
</tr>
<tr>
<td>Processor evolution</td>
</tr>
<tr>
<td>Memory Caches</td>
</tr>
<tr>
<td>Mixing code and work areas: a poor practice</td>
</tr>
<tr>
<td>Interlocks</td>
</tr>
<tr>
<td>Incrementing addresses</td>
</tr>
<tr>
<td>Guidelines for Part 3</td>
</tr>
<tr>
<td>**Part 4: Improving program structure and maintainability**</td>
</tr>
<tr>
<td>CEJECT for improved listing readability</td>
</tr>
<tr>
<td>The incredibly useful and powerful LOCTR assembler instruction</td>
</tr>
<tr>
<td>Minimizing base register requirements</td>
</tr>
<tr>
<td>The HLASM Toolkit’s Structured Programming Macros</td>
</tr>
<tr>
<td>Advice from experienced (and very successful!) programmers</td>
</tr>
</tbody>
</table>
## Table of Contents

**Summary** .................................................................................................................. 56
- Things worth remembering .................................................................................... 57
- Subscribing to ASSEMBLER and IBM-MAIN Discussion Groups .................. 58
- Useful references .................................................................................................. 59
• Part 1: Tidying up portions of your programs
  – Easy changes that can make small segments of code more manageable

• Part 2: Upgrading instructions to newer, efficient forms
  – Simple instructions that can make code clearer, smaller, and more efficient

• Part 3: Enhancing awareness of CPU behavior
  – Little things that can make critical sequences more efficient

• Part 4: Improving readability and maintainability
  – Ways you can clarify and simplify program organization

• Summary observations
Part 1: Tidying up your programs

Assembler Language doesn't have to be difficult!
Counting Characters

• A common instruction sequence:

\[
\text{MVC Buffer(74),=C'\text{Message of about 74 (\?\?) characters...'}}
\]

  – Problem: Some poor soul (you?) had to count the characters to get the “74”
    — Or, didn't want to count, and decided 74 was more than long enough

• Better: define a constant containing the message:

\[
\text{MVC Buffer(L'\text{Msg3}),Msg3}
\]

\[
\text{Msg3 DC C'\text{Message of (I don't care how many) characters...'}}
\]

• Advantages:
  – The assembler counts the number of characters (correctly!)
  – 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
Counting characters vs. a simple “MVC2” macro

- If modifying code to use Length Attributes is too tedious, use a **MVC2** macro:

  MVC Buffer(74),=CL74'Message of < than 74 chars' Old way
  MVC2 Buffer,=C'Message of any number of chars' New way

- Ask someone to install this macro in your macro library:

  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 generated instruction is

  MVC Target(L'&Source),&Source Just what you wanted!
  MVC2 Buffer,=C'A long message...' MVC2 handles everything

  - It automatically uses the length attribute of the second operand
A common instruction sequence

\[
\begin{align*}
\text{MVI} & \quad \text{Buffer}, \text{C'} ' \quad \text{Clear a buffer to blanks} \\
\text{MVC} & \quad \text{Buffer}+1(132), \text{Buffer} \quad \text{Ripple the first blank} \\
\end{align*}
\]

\[
\text{Buffer} \quad \text{DS} \quad \text{CL133}
\]

- Problem: what if the length of the buffer must be changed?
- You must find all occurrences of the symbol \text{Buffer} and change 132, 133 (and maybe other numbers)

Better:

\[
\begin{align*}
\text{BufLen} & \quad \text{Equ} \quad 133 \quad \text{Define the buffer length} \\
\text{MVI} & \quad \text{Buffer}, \text{C'} ' \quad \text{Clear a buffer to blanks} \\
\text{MVC} & \quad \text{Buffer}+1(\text{BufLen}-1), \text{Buffer} \quad \text{Ripple the first blank} \\
\end{align*}
\]

\[
\text{Buffer} \quad \text{DS} \quad \text{CL(Buflen)}
\]

- Advantage: you need to change only the statement defining \text{BufLen}, and reassemble
- Instructions don’t need to know anything about data declarations
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 had to count 102 bytes!
  DC 17F'0' (should be 18!)  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

- Better:

  &Name Start
  J S&SysNdx The Assembler knows where to go:
  DC 18F'0' Corrected!
  DC C'Assembled &SysDatC'
  DC C'Time &SysTime'
  DC C'At Site &ThisLoc.' ... Additional
  DC C'With HLASM &SysVer.' ... signature
  DC C'on System &System_ID.' ... information
  S&SysNdx STM R14,R12,12(R13)
Two statement sequences to define a record and its fields:

```
record  DS OCL923 923?
record  DS OCL(RecLen)
Entry   DC H'923' 923??
Entry   DS Y(RecLen)
Field1  DS CL44
Field1  DS CL44
Field2  DS CL55
Field2  DS CL55
Field999 DS ...
```

---

Problems:
1. Someone counted the Fieldnn lengths to determine “923” (risky!)
2. HLASM complains about the apparently better symbolic definition

A better method: let the Assembler do all the work 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
RecLen    Org RecHead  Re–position at start of record
Record    DS OCL(RecLen) Define name and length of entire record
Record    Org ,        Re–position after the record
```
A typical instruction sequence to add inserts in a message:

\[
\begin{align*}
&MVC \quad Buffer+64(12),\text{Insert1} \quad \text{Insert something somewhere} \\
&MVC \quad Buffer+82(10),\text{Insert2} \quad \text{Insert something somewhere}
\end{align*}
\]

- Problem: if the report must be reformatted, you have to look for all the offsets and lengths

- Better: define the insertion points where the Buffer is defined

\[
\begin{align*}
&\text{Buffer} \quad DS \quad CL133 \\
&\text{Insert1} \quad DS \quad CL12 \quad \text{Inserted data} \\
&\text{Insert2} \quad DS \quad CL10 \quad \text{Inserted data}
\end{align*}
\]

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

• Still better: 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  CL12
  Insert2 DS  CL10

  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:
  – No explicit lengths or offsets in the MVC instructions
  – Changes localized to the DSECT
  – Instructions don't need to know anything about data declarations
Duplicated record definitions

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

<table>
<thead>
<tr>
<th>New Record Declaration</th>
<th>Old Record Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewRec DS OD</td>
<td>OldRec DS OD</td>
</tr>
<tr>
<td>NewType DS CL10 Record type</td>
<td>OldType DS CL10</td>
</tr>
<tr>
<td>NewID DS CL4 Record ID</td>
<td>OldID DS CL4</td>
</tr>
<tr>
<td>NewName DS CL40 Name</td>
<td>OldName DS CL40</td>
</tr>
<tr>
<td>NewAddr DS CL66 Address</td>
<td>OldAddr DS CL66</td>
</tr>
<tr>
<td>NewPhone DS CL12 Phone number</td>
<td>OldPhone DS CL12</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLC NewID,OldID</td>
</tr>
</tbody>
</table>

- Everything addressed by current base register(s)
- Big, BIG trouble if the declarations get out of sync
Better: define a *single* DSECT describing the record

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record</td>
<td>DSECT</td>
<td>Record description</td>
</tr>
<tr>
<td>RecType</td>
<td>DS</td>
<td>Record type</td>
</tr>
<tr>
<td>RecID</td>
<td>DS</td>
<td>Record ID</td>
</tr>
<tr>
<td>RecName</td>
<td>DS</td>
<td>Name</td>
</tr>
<tr>
<td>RecAddr</td>
<td>DS</td>
<td>Address</td>
</tr>
<tr>
<td>RecPhone</td>
<td>DS</td>
<td>Phone number</td>
</tr>
<tr>
<td>RecYear</td>
<td>DS F</td>
<td>Processing year</td>
</tr>
<tr>
<td>RecDay</td>
<td>DS F</td>
<td>Processing day of year</td>
</tr>
<tr>
<td>RecLen</td>
<td>Equ</td>
<td>*-Record</td>
</tr>
<tr>
<td>NewRec</td>
<td>DS</td>
<td>OD,CL(RecLen)</td>
</tr>
<tr>
<td>OldRec</td>
<td>DS</td>
<td>OD,CL(RecLen)</td>
</tr>
</tbody>
</table>

- Advantage: everyone can use the same record definition
  - It can be in a COPY segment or generated by a macro

- Next two slides show how to utilize the **Record** DSECT
Enhanced USING Statements

1. With separate base registers for code and for each record instance:

   - Labeled USING statements; qualifiers are **Old** and **New**

   ```assembly
   MyProg CSECT ,
   _ _ _
   LA 7,OldRec Base register for OldRec
   LA 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
2. With *existing* base registers for code *and* each record instance

- Labeled Dependent USING statements; 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 “anchored” on each record field
- Program base register(s) address everything
- This version uses exactly the same base registers as the original
Unreferenced code and data

- Stuff tends to accumulate even when it's no longer needed
  - Problem: the next person may not be sure something is not needed, so leaves it untouched
  - Worse: a statement label in dead code could be an inviting branch target
- Solution: specify Assembler option \texttt{XREF(SHORT,UNREFS)} (the default)

\begin{verbatim}
Unreferenced Symbols Defined in CSECTs

\begin{tabular}{ll}
\text{Defn} & \text{Symbol} \\
\hline
 & \\
 & \\
674 & ADDCOM \quad \text{← The unreferenced symbol and the} \\
724 & ADDDIM \quad \text{statement where it's defined} \\
1011 & AUTORT \\
860 & BLANKS \\
630 & CKDIM \\
1038 & CLOSE11 \\
 & \\
\end{tabular}
\end{verbatim}

- If you don't want to delete the statements, skip them:
  \begin{verbatim}
AGo .Skip04 \quad \text{Skip the leftovers}
--- \quad \text{Unreferenced odds and ends}
.Skip04 ANop , \quad \text{Intervening statements not assembled}
\end{verbatim}

- \textbf{Don't} hide unused statements following the END statement!
Register equates and “names”

• Many programs contain EQU statements to “name” registers

\[
\begin{align*}
R0 & \text{ Equ } 0 \\
R15 & \text{ Equ } 15
\end{align*}
\]

– ... in the belief that doing so helps you find references in the Symbol XREF
– Unfortunately, this isn’t true:

\[
\text{LM } R14, R12, 0(R13) \quad \text{Refers to all 16 general registers!}
\]
– Only \textbf{R12}, \textbf{R13}, and \textbf{R14} will appear in the XREF

• Much better: rely on the \textbf{Register} XREF (specify the RXREF option)

• Another problem: beginners may think register “names” are reserved (as on Intel processors), and write

\[
\text{L } R5, R8 \quad \text{Load Register 8 into Register 5 (??)}
\]

• Usually safest just to use register numbers
  – If your code uses general, floating-point, and access registers: names might help clarify which is which (but \textit{not} with implicit references)
Part 2: Benefiting from newer, efficient instructions

- Nifty new easy-to-use instructions
  - Reduce the costs of memory references
Topics

• Quick review of some z/Architecture features

• Instructions with immediate operands
  – Load and insert instructions
  – Arithmetic instructions
  – Logical instructions

• Address Generation
  – Base and unsigned 12-bit displacement
  – Base and signed 20-bit displacement
  – Instruction-relative addressing

• Relative addressing

• Other instructions worth knowing about
• We're familiar with SI-type instructions with “immediate” operands:

<table>
<thead>
<tr>
<th>opcode</th>
<th>$I_2$</th>
<th>$B_1$</th>
<th>$D_1$</th>
</tr>
</thead>
</table>

  – Used for instructions with logical operands, like MVI, CLI, TM, OI, etc.

• Newer instructions with immediate-operand support
  – Arithmetic (signed and unsigned)
  – Logical operations (up to 32 bits)
  – Branch relative (no base register required!)

• Greater flexibility, many different types of operand
• 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:

\[
\begin{array}{cccccc}
\text{HH} & \text{HL} & \text{LH} & \text{LL} \\
0 & 15 & 16 & 31 & 32 & 47 & 48 & 63
\end{array}
\]

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

<table>
<thead>
<tr>
<th>HH</th>
<th>High Half's High Half (bits 0-15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HL</td>
<td>High Half's Low Half (bits 16-31)</td>
</tr>
<tr>
<td>LH</td>
<td>Low Half's High Half (bits 32-47)</td>
</tr>
<tr>
<td>LL</td>
<td>Low Half's Low Half (bits 48-63)</td>
</tr>
<tr>
<td>H</td>
<td>High Half (bits 0-31)</td>
</tr>
<tr>
<td>L</td>
<td>Low Half (bits 32-63)</td>
</tr>
</tbody>
</table>
## 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand 1</th>
<th>32-bit register</th>
<th>64-bit register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Operand 2</td>
<td>16 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Arithmetic Load</td>
<td>LHI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Load</td>
<td></td>
<td></td>
<td>LLIHH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LLIHL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LLIILH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LLILL</td>
</tr>
<tr>
<td>Insert</td>
<td>IILH</td>
<td>IILF</td>
<td>IIHH</td>
</tr>
<tr>
<td></td>
<td>IILL</td>
<td></td>
<td>IIHL</td>
</tr>
</tbody>
</table>
### Examples using load-immediate instructions

<table>
<thead>
<tr>
<th>Old Ways</th>
<th>Better Ways</th>
</tr>
</thead>
<tbody>
<tr>
<td>L 1,=F'275'</td>
<td>LHI 1,275</td>
</tr>
<tr>
<td>LH 2,=H'─5678'</td>
<td>LHI 2,─5678</td>
</tr>
<tr>
<td>L 3,=F'123456789'</td>
<td>LGFI 3,123456789 (64-bit register)</td>
</tr>
<tr>
<td></td>
<td>IILF 3,123456789 (32-bit register)</td>
</tr>
</tbody>
</table>

- Eliminate memory references and constants in storage
- Eliminate unnecessary register zeroing, needless memory references
- Faster operation, smaller programs, no base register needed
### Arithmetic instructions with immediate operands

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand 1</th>
<th>32-bit register</th>
<th>64-bit register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>16 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Arithmetic Add/Subtract</td>
<td>AHI</td>
<td>AFI</td>
<td>AGHI</td>
</tr>
<tr>
<td>Logical Add/Subtract</td>
<td></td>
<td>ALFI, SLFI</td>
<td>ALGFI, SLGFI</td>
</tr>
<tr>
<td>Arithmetic Compare</td>
<td>CHI</td>
<td>CFI, CRL</td>
<td>CGHI</td>
</tr>
<tr>
<td>Logical Compare</td>
<td></td>
<td>CLFI</td>
<td>CLGFI</td>
</tr>
<tr>
<td>Multiply</td>
<td>MHI</td>
<td></td>
<td>MGHI</td>
</tr>
</tbody>
</table>

- Instructions referencing 32-bit registers are immediately useful

<table>
<thead>
<tr>
<th>Old Ways</th>
<th>Better Ways</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 6,=A(Offset*4)</td>
<td>AFI 6,Offset*4</td>
</tr>
<tr>
<td>CH 4,=H'─1'</td>
<td>CHI 4,─1</td>
</tr>
<tr>
<td>MH 2,=Y(ItemLen)</td>
<td>MHI 2,ItemLen</td>
</tr>
<tr>
<td>CL 9,=X'107429B3'</td>
<td>CLFI 9,X'107429B3'</td>
</tr>
</tbody>
</table>

- Faster operation, smaller programs, no base register needed
• Instructions operate on 32 bits of a 64-bit register, or on 16-bit high or low halves of each half

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand 1</th>
<th>64-bit register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Operand 2</td>
<td>16-bit immediate operand</td>
</tr>
<tr>
<td>AND</td>
<td>NIHH, NIHL NILH, NILL</td>
<td>NIHF NILF</td>
</tr>
<tr>
<td>OR</td>
<td>OIHH, OIHL OILH, OILL</td>
<td>OIHF OILF</td>
</tr>
<tr>
<td>XOR</td>
<td></td>
<td>XIHF XILF</td>
</tr>
<tr>
<td>Test Under Mask</td>
<td>TMHH, TMHL TMLH, TMLL</td>
<td></td>
</tr>
</tbody>
</table>

• Underscored instructions operate within the rightmost 32 bits
  – Exercise for the reader: why are the AND and OR instructions with 16-bit operands unnecessary?
• Isolate the rightmost 6 bits of GR4:

<table>
<thead>
<tr>
<th>Old Ways</th>
<th>Better Way</th>
</tr>
</thead>
<tbody>
<tr>
<td>N 4=X'0000003F'</td>
<td>NILL 4,X'3F'</td>
</tr>
<tr>
<td>SLL 4,26</td>
<td>NILL 4,X'3F'</td>
</tr>
<tr>
<td>SRL 4,26</td>
<td>NILL 4,X'3F'</td>
</tr>
<tr>
<td>SRDL 4,6 (lose R5 bits!)</td>
<td>NILL 4,X'3F'</td>
</tr>
<tr>
<td>SR 4,4</td>
<td></td>
</tr>
<tr>
<td>SLDL 4,6</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Old Way</th>
<th>Better Way</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR 0,5</td>
<td>TMLH 5,X'7F00'</td>
</tr>
<tr>
<td>SLL 0,1</td>
<td>JZ Its_Safe</td>
</tr>
<tr>
<td>SRA 0,25</td>
<td></td>
</tr>
<tr>
<td>JZ Its_Safe</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Old Way</th>
<th>Better Way</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR 0,9</td>
<td>TMLL 9,X'0003'</td>
</tr>
<tr>
<td>N 0=X'3'</td>
<td>JZ Mult4</td>
</tr>
<tr>
<td>JZ Mult4</td>
<td></td>
</tr>
</tbody>
</table>

• In each case: extra register, extra instructions, or memory reference
Review of base-displacement address generation

1. With unsigned 12-bit displacement

<table>
<thead>
<tr>
<th>base b</th>
<th>displacement</th>
</tr>
</thead>
</table>

   • 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:

<table>
<thead>
<tr>
<th>opcode</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>opcode</th>
</tr>
</thead>
</table>

   • Traditional unsigned 12-bit displacement field now named DL₂
   • High-order 8-bit signed displacement extension named DH₂
26 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

```
| instruction | b | DL | DH |
```

- Displacement range \((-2^{19}, +2^{19} - 1)\) rather than \((0,4095)\)

- Address calculation adds base/index register contents as appropriate
  - Number of significant digits depends on current addressing mode

- If the DH field is zero, get usual 12-bit displacement
• 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!
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>R:AB 000000</td>
<td>00012</td>
<td>1</td>
<td>Test CSECT, Using *,10,11</td>
</tr>
<tr>
<td>000000</td>
<td>X Equ *</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>E300 B880 1208</td>
<td>13880</td>
<td>4</td>
<td>AG 0,X+80000 Long displacement</td>
</tr>
<tr>
<td>000006</td>
<td>E300 AFA0 0008</td>
<td>00FA0</td>
<td>5</td>
<td>AG 0,X+4000 R11 +96 bytes away</td>
</tr>
<tr>
<td>00000C</td>
<td>E300 BFA0 FF08</td>
<td>00FA0</td>
<td>6</td>
<td>Drop 10</td>
</tr>
<tr>
<td>000012</td>
<td>E300 0120 7A71</td>
<td>7A120</td>
<td>7</td>
<td>AG 0,X+4000 Negative displacement</td>
</tr>
<tr>
<td>000018</td>
<td>E300 0EE0 8371</td>
<td>F83000</td>
<td>8</td>
<td>* Note absolute displacements:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9  LAY 0,+500000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 LAY 0,−512000 AMode sensitive!</td>
</tr>
</tbody>
</table>

- DH fields are underscored
• New instruction formats with 2-byte and 4-byte immediate operands

• 4-byte instruction:

<table>
<thead>
<tr>
<th>opcode</th>
<th>R₁</th>
<th>op</th>
<th>RL₂</th>
</tr>
</thead>
</table>

RL₂ range: $-2^{15} \leq I₂ \leq 2^{15} - 1$, or $-32768 \leq I₂ \leq 32767$

• 6-byte instruction:

<table>
<thead>
<tr>
<th>opcode</th>
<th>R₁</th>
<th>op</th>
<th>RL₂</th>
</tr>
</thead>
</table>

RL₂ range: $-2^{31} \leq RI₂ \leq 2^{31} - 1$, or $-2147483648 \leq RI₂ \leq 2147483647$
Relative addressing ...

- **Address generation:**

  ![Diagram](image)

  - **RI**-type instruction
  - Shift left 1 bit
  - 64-bit signed offset
  - (Not the IA in the PSW!)
  - **Effective Address**

- **RI₂** operand is doubled because a branch target is always on an even boundary

- **No** base register required; base register requirement(s) can be minimized
Important relative branch instructions

• Branch Relative on Condition:

\[
\begin{array}{cccc}
A7 & M_1 & 4 & RI_2 \\
\end{array}
\]

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

• Branch Relative Long on Condition:

\[
\begin{array}{cccc}
C0 & M_1 & 4 & RI_2 \\
\end{array}
\]

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?)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Immediate-Operand Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch on Condition (Relative)</td>
<td>( BC R ) [JC]</td>
</tr>
<tr>
<td>Branch on Condition (Relative)</td>
<td>( BC RL ) [JLC]</td>
</tr>
</tbody>
</table>

• Extended mnemonics in [square brackets] start with J (for “Jump”)
### Relative branch on condition instructions and extended mnemonics

<table>
<thead>
<tr>
<th>RI Mnemonics</th>
<th>RIL Mnemonics</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRC</td>
<td>JC</td>
<td></td>
<td>Conditional Branch</td>
</tr>
<tr>
<td>BRU</td>
<td>J</td>
<td>15</td>
<td>Unconditional Branch</td>
</tr>
<tr>
<td>BRNO</td>
<td>JO</td>
<td>14</td>
<td>Branch if Not Ones (T)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Branch if No Overflow (A)</td>
</tr>
<tr>
<td>BRNH</td>
<td>JNH</td>
<td>13</td>
<td>Branch if Not High (C)</td>
</tr>
<tr>
<td>BRNP</td>
<td>JNP</td>
<td>13</td>
<td>Branch if Not Plus (A)</td>
</tr>
<tr>
<td>BRNL</td>
<td>JNL</td>
<td>11</td>
<td>Branch if Not Low (C)</td>
</tr>
<tr>
<td>BRNM</td>
<td>JNM</td>
<td>11</td>
<td>Branch if Not Minus (A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Branch if Not Mixed (T)</td>
</tr>
<tr>
<td>BRE</td>
<td>JE</td>
<td>8</td>
<td>Branch if Equal (C)</td>
</tr>
<tr>
<td>BRZ</td>
<td>JZ</td>
<td>8</td>
<td>Branch if Zero(s) (A,T)</td>
</tr>
<tr>
<td>BRNZ</td>
<td>JNZ</td>
<td>7</td>
<td>Branch if Not Equal (C)</td>
</tr>
<tr>
<td>BRNE</td>
<td>JNE</td>
<td>7</td>
<td>Branch if Not Zero (A,T)</td>
</tr>
<tr>
<td>BRL</td>
<td>JL</td>
<td>4</td>
<td>Branch if Low (C)</td>
</tr>
<tr>
<td>BRM</td>
<td>JM</td>
<td>4</td>
<td>Branch if Minus (A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Branch if Mixed (T)</td>
</tr>
<tr>
<td>BRH</td>
<td>JH</td>
<td>2</td>
<td>Branch if High (C)</td>
</tr>
<tr>
<td>BRP</td>
<td>JP</td>
<td>2</td>
<td>Branch if Plus (A)</td>
</tr>
<tr>
<td>BRO</td>
<td>JO</td>
<td>1</td>
<td>Branch if Ones (T)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Branch if Overflow (A)</td>
</tr>
<tr>
<td>JNOP</td>
<td>JLNOP</td>
<td>0</td>
<td>No Operation</td>
</tr>
</tbody>
</table>

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

- Loop control instructions:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Register Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 bits</td>
</tr>
<tr>
<td>Branch on Count (Register)</td>
<td>BCTR</td>
</tr>
<tr>
<td>Branch on Count (Indexed)</td>
<td>BCT</td>
</tr>
<tr>
<td>Branch on Count (Relative)</td>
<td>BRCT [JCT]</td>
</tr>
<tr>
<td>Branch on Index</td>
<td>BXH</td>
</tr>
<tr>
<td></td>
<td>BXLE</td>
</tr>
<tr>
<td>Branch on Index (Relative)</td>
<td>BRXH [JXH]</td>
</tr>
<tr>
<td></td>
<td>BRXLE [JXLE]</td>
</tr>
</tbody>
</table>

**EXRL** Execute Relative Long: no base register required

**LARL** Load Address Relative Long: no base register required (target must be an even address)
Other useful relative-address instructions ...

- Branch and save instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Immediate-Operand Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16 bits</td>
</tr>
<tr>
<td>Branch and Save (Relative)</td>
<td>BRAS [JAS]</td>
</tr>
</tbody>
</table>

- 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
Compare and branch instructions combine the two operations:

<table>
<thead>
<tr>
<th>CRB, CGRB</th>
<th>Compare and Branch</th>
<th>CRJ, CGRJ</th>
<th>Compare and Branch Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIB, CGIB</td>
<td>Compare Immediate and Branch</td>
<td>CIJ, CGIJ</td>
<td>Compare Immediate and Branch Relative</td>
</tr>
</tbody>
</table>

- The I₂ (comparand) operand is a signed 8-bit number
- All instructions support extended mnemonics

<table>
<thead>
<tr>
<th>Old Ways</th>
<th>Better Ways</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR 3,4</td>
<td>CRJNE 3,4,NotSame</td>
</tr>
<tr>
<td>JNE NotSame</td>
<td></td>
</tr>
<tr>
<td>C 9,=F'–99'</td>
<td>CIJL 9,–99,TooSmall</td>
</tr>
<tr>
<td>JL TooSmall</td>
<td></td>
</tr>
<tr>
<td>LTR 0,0</td>
<td>CIJNM 0,0,NotMinus</td>
</tr>
<tr>
<td>JNM NotMinus</td>
<td></td>
</tr>
</tbody>
</table>

- CRB, CGRB, CIB, and CGIB are based branches
- Save (several) instructions and memory references
• Relative branches may have eliminated the need for base registers for your code, but...

• Many IBM macros generate based instructions like 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
Conditional load and store instructions

- Load or store action depends on Condition Code setting

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 bits</td>
</tr>
<tr>
<td>Load Register</td>
<td>LROC</td>
</tr>
<tr>
<td>Load</td>
<td>LOC</td>
</tr>
<tr>
<td>Store</td>
<td>STOC</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Old Way</th>
<th>New Way</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR 2,0</td>
<td>LR 2,0</td>
</tr>
<tr>
<td>CR 0,1</td>
<td>CR 0,1</td>
</tr>
<tr>
<td>JNL OK</td>
<td>LROCL 2,1</td>
</tr>
<tr>
<td>LR 2,1</td>
<td>No, C(R0)&lt;c(R1)</td>
</tr>
<tr>
<td>OK</td>
<td>Load if c(R0)&lt;c(R1)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL</td>
<td>2,12</td>
<td>Original contents of R2 changed</td>
</tr>
<tr>
<td>AR</td>
<td>4,7</td>
<td>Original contents of R4 changed</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLLK</td>
<td>0,2,12</td>
<td>Result in R0; contents of R2 unchanged</td>
</tr>
<tr>
<td>ARK</td>
<td>3,4,7</td>
<td>Sum in R3; contents of R4 unchanged</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Old Way</th>
<th>New Way</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>ARK</td>
</tr>
<tr>
<td>3,4</td>
<td>3,4,7</td>
</tr>
<tr>
<td>AR</td>
<td></td>
</tr>
<tr>
<td>3,7</td>
<td></td>
</tr>
</tbody>
</table>
Part 3: Enhancing awareness of CPU behavior

• These items can be important for CPU-intensive or frequently-executed programs
• 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 overlap each of those three steps (and split them into many additional stages) in a “pipeline”
  ─ Anything that affects pipeline flow will slow execution
  ─ There are many conditions that affect performance at the instruction level
• It's important to understand how your code can affect cache behavior
• 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
• 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 OH
MVC Somewhere(L'Temp),Temp Move the result
```

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

<table>
<thead>
<tr>
<th>Old Way</th>
<th>New Way</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST 2,Result</td>
<td>ST 2,Result</td>
</tr>
<tr>
<td>CLC Result,OldValue</td>
<td>CL 2,OldValue</td>
</tr>
<tr>
<td>MVC WorkArea(8),Data</td>
<td>MVC WorkArea(8),Data</td>
</tr>
<tr>
<td>CLI WorkArea+7,C'A'</td>
<td>CLI Data+7,C'A'</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>LTR 15,15</th>
<th>Check for error</th>
</tr>
</thead>
<tbody>
<tr>
<td>JNZ Error_27</td>
<td>Branch only on unusual condition</td>
</tr>
<tr>
<td>---</td>
<td>Continue normal processing</td>
</tr>
</tbody>
</table>
Incrementing addresses

- AHI vs. LA

\[
\begin{align*}
L & 0,0(,6) & \text{Load GRO} \\
LA & 6,4(,6) & \text{Increment pointer}
\end{align*}
\]

may be slightly faster than

\[
\begin{align*}
L & 0,0(,6) & \text{Load GRO} \\
AHI & 6,4 & \text{Increment pointer}
\end{align*}
\]

- For address incrementation, it's usually better to use LA rather than AHI
  - Special hardware for expediting LA

- Be very careful if you use LA, LAY for arithmetic: the results depend on the current addressing mode
• 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 data frequently read (but infrequently updated) separate from data frequently updated
  – Keep serialized objects on separate cache lines

• Keep referenced data close in memory and in time

• 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
Part 4: Improving program structure and maintainability

- Ways to cope with ever-expanding programs
• Listings don't always keep related chunks of code together

• Use CEJECT (“Conditional Eject”) to keep them grouped

  CEject 12
  ├── ─ ─ ─ ┐ │ ─ ─ ─ │ 12 statements kept on one page └─ ─ ─ ─ ┘

  CEject 5
  ├── ─ ─ ─ ┐ │ ─ ─ ─ │ 5 statements kept on one page └─ ─ ─ ─ ┘

  – CEJECT counts lines remaining on the page, ejects if not enough

• Improved readability improves understanding
The incredibly useful and powerful LOCTR assembler instruction

- LOCTR keeps groups of related statements together in the source code
  - 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
         c...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
         LTORG , Your 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
```

- HLASM sorts the groups in order of declaration, so the object code looks like:

```
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
```
• Example:

```
Code    LOCTR,
   MVC  WorkBuff(L'Message5),Message5 Move message to buffer
Messages LOCTR,
Message5 DC  C'What can you possibly be doing?'
WorkArea LOCTR,
WorkBuff DS  CL(BuffLen) Define the message buffer
Code    LOCTR,
   LAY 0,L'Message5 Set up length for write subroutine
   LAY 1,Message5 Set address for write subroutine
   JAS 14,MsgWrite Call message–writer subroutine
   OI  BugBit,L'BugBit Set a flag indicating this error
WorkArea LOCTR,
   DS  X Define a byte for some flag bits
BugBit  Equ  *-1,X'40' Define the error–indicator flag bit
EOFBit  Equ  *-1,X'08' Define an end-of-file flag bit...
```

• Shows how you can keep related statements together in the source file
Minimizing base register requirements

1. Use LOCTR to group related items at the start of the CSECT

2. Use only relative branches among instructions in the program area
   - Use EXRL for any EXecute instructions in the code
   - So there's no need for "code base" registers

3. When appropriate, use LAY and LARL to reference constants, literals, and work area items

Base register(s) are needed only for constants, literals, and the work area!
The HLASM Toolkit’s Structured Programming Macros

- Powerful tools for improving program structure
- Provide uniformity and standardization
- Reduce the number of different constructs used in a program
- Better tools for thinking about programs

- Enhance program readability and maintainability
  - Eliminate GOTO statements, extraneous labels, out-of-line logic paths
    - Statement labels represent “unstructured” exposures; each label is a tempting branch target
  - Easier to understand program flow without tedious inspection
    - Far less “spaghetti code”
  - Some users report SP macros reduce maintenance costs by over 50%

- No more effort to use the SP macros than in a 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
• Converting unstructured code
  – You can mix structured and unstructured code
    — Start small, and work from the “inside out’
    — If-Then-Else, Do-EndDo are very easy to get started with
    — Small changes are quick and easy; programs gradually gain structure
  – Add structure incrementally, leave old code alone if it’s too much bother
    — “Spaghetti code” is harder to restructure
  – Use constructs that will make it easy to add new cases in the future
  – Major rewrites or new programs represent structuring opportunities
  – You can get rid of almost all statement labels: a good thing!

• Remember! Conversion is never required!

• You can customize the macro names to local standards by editing the ASMMNAME copy file
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
Summary
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 nearby instructions is very expensive (especially if they are executed repeatedly)
4. Group constants and literals together; same for work areas
   • Locality of reference helps performance
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 figure out what you did to fix it
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@UA1VM.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


- “User Experience – Tuning Old Assembler Code to Exploit z/Architecture”. SHARE Feb. 2007, Session 8185

- “A ‘Quick Start’ Approach to Training Anyone to Write Assembler Language”. SHARE Mar. 2009, Session 8144


- “Reducing Base Register Utilization: How to ‘Jumpify’ Your Programs”. SHARE Feb. 2011, Session 8548

- “How to Benefit From HLASM's Most Powerful Features”. SHARE Aug. 2011, Session 9223

- IBM documentation:
  - High Level Assembler Language Reference SC26-4940
  - High Level Assembler Programmer's Guide SC26-4941
  - z/Architecture Principles of Operation SA22-7832
  - z/Architecture Reference Summary SA22-7871