Upgrading Assembler Language Programs:

Tips and Techniques

SHARE 117, Session 9281

John R. Ehrman
ehrman@us.ibm.com

IBM Silicon Valley (nee Santa Teresa) Laboratory
555 Bailey Avenue
San Jose, CA 95141

August 10, 2011

- 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!

- A common instruction sequence:

```
        MVC   Buffer(74),=C'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:

```
        MVC   Buffer(L'Msg3),Msg3
        - - -
Msg3    DC    C'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

- 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

```
          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)

- Better:

```
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

- 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:

```
ARecord  DS    OCL923  923?
RecHead  DC    H'923'  923??
Field1   DS    CL44
Field2   DS    CL55
         — — —
Field999 DS    ...
```

```
   ARecord  DS   OCL(RecLen)
   RecHead  DS   Y(RecLen)
   Field1   DS   CL44
            — — —
   RecLen   Equ  *—ARecord
** ASMA080E Statement is unresolvable (!)
```

  - 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
         Org   RecHead        Re—position at start of record
ARecord  DS    OCL(RecLen)    Define name and length of entire record
         Org   ,              Re—position after the record
```

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

```
          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: 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

```
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

- 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

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

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

         CLC   NewID,OldID        Compare record IDs (we hope!)
```

- 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

```
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)      Area for new record
OldRec   DS    0D,CL(RecLen)      Area for old record
```

- 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

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

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

```
MyProg   CSECT ,                        Resume program control section
         — — —
         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

- 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 **XREF(SHORT,UNREFS)** (the default)

```
                                Unreferenced Symbols Defined in CSECTs
Defn  Symbol
 ...
 674  ADDCOM                    <-- The 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!

- Many programs contain EQU statements to “name” registers

```
R0      Equ   0
        - - -
R15     Equ   15
```

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

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

  - Only **R12**, **R13**, and **R14** will appear in the XREF

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

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

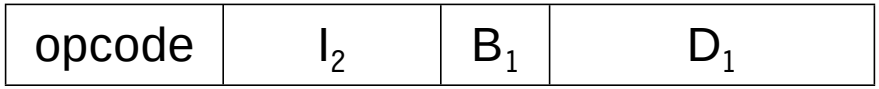
- Usually safest just to use register numbers
  - If your code uses general, floating-point, and access registers: names might help clarify which is which (but *not* with implicit references)

Part 2: Benefiting from newer, efficient instructions

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

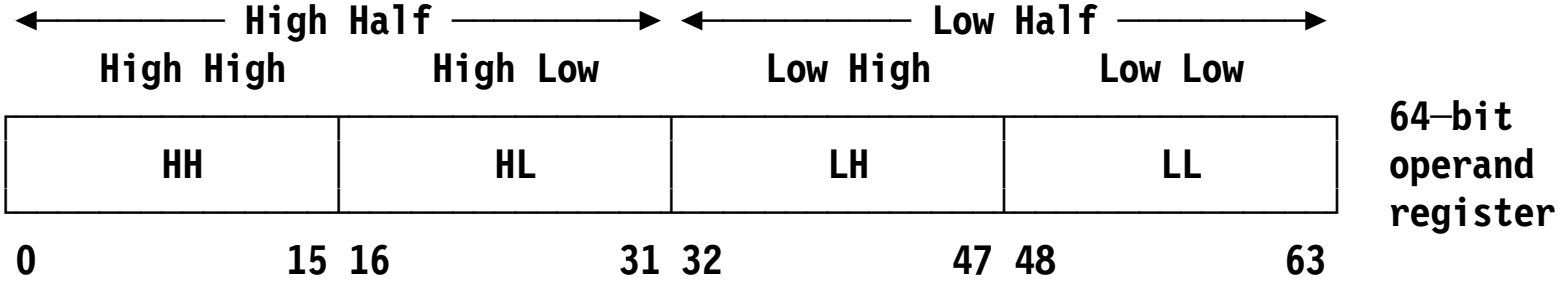
- Quick review of some z/Architecture features
- Instructions with immediate operands
  - Load and insert instructions
  - Arithmetic instructions
  - Logical instructions
- Address Generation
  - Base and unsigned 12-bit displacement
  - Base and signed 20-bit displacement
  - Instruction-relative addressing
- Relative addressing
- Other instructions worth knowing about

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



  - 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

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



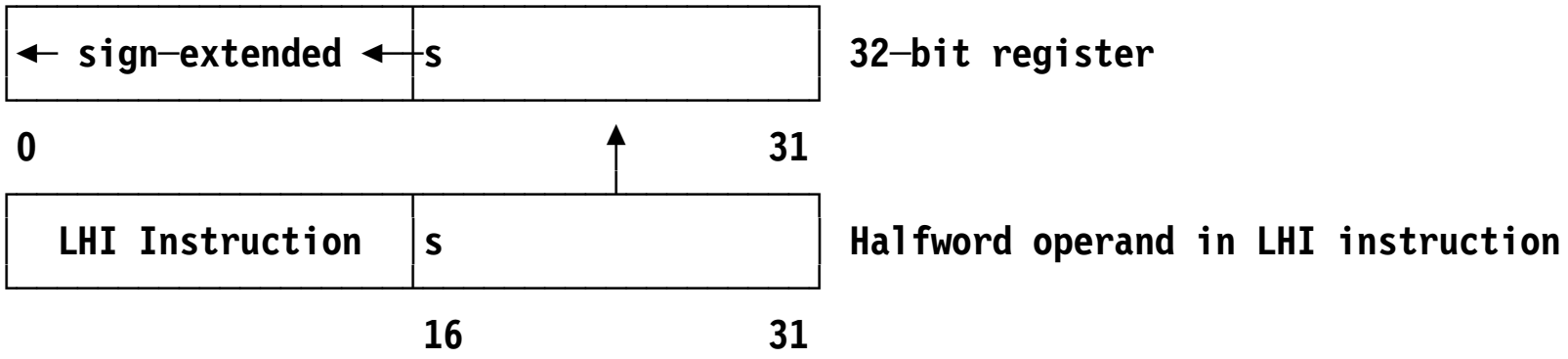
- Last one or two letters of many instruction mnemonics indicate which part of the GR is involved:

HH High Half's High Half (bits 0-15)
HL High Half's Low Half (bits 16-31)
LH Low Half's High Half (bits 32-47)
LL Low Half's Low Half (bits 48-63)

H High Half (bits 0-31)
L Low Half (bits 32-63)

- 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



- 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

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

- 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?

- 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?

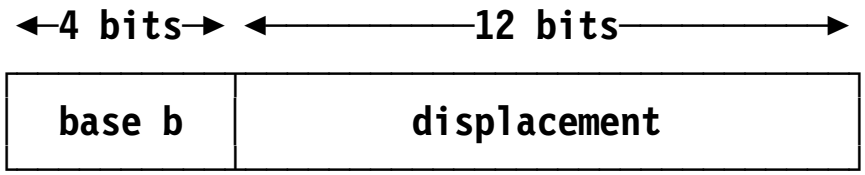
```
Old Way                           Better Way
LR    0,5                         TMLH  5,X'7F00'
SLL   0,1                         JZ    Its_Safe
SRA   0,25
JZ    Its_Safe
```

- 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, extra instructions, or memory reference

1. With unsigned 12-bit displacement



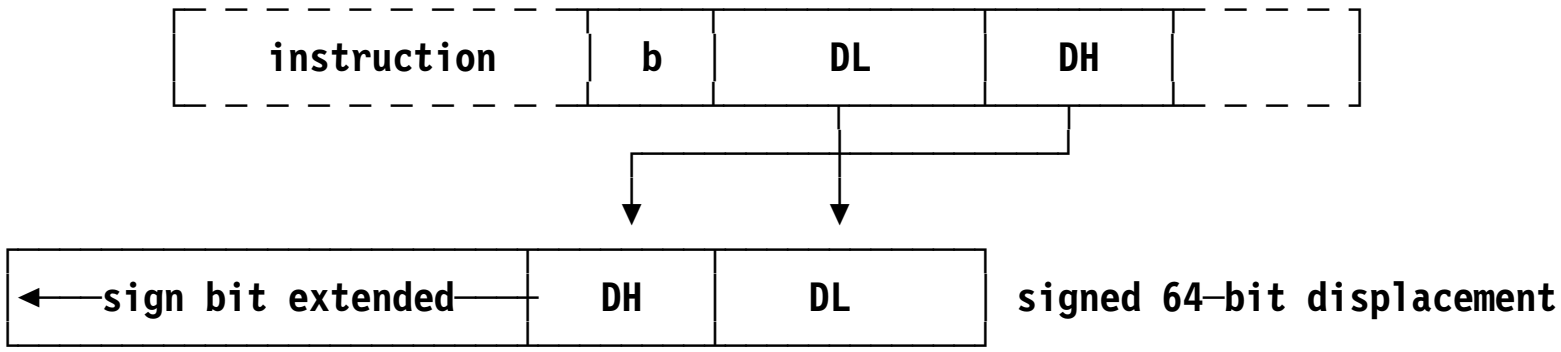
- Effective Address = displacement + [***if*** (b ≠ 0) ***then*** c(GRb)]
- Provides addressability to at most 4096 bytes per base register
  - And, you can't address anything preceding the generated address

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

opcode	R_1	X_2	B_2	DL_2	DH_2	opcode

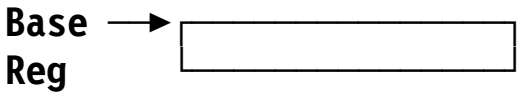
- Traditional unsigned 12-bit displacement field now named **DL_2**
- High-order 8-bit signed displacement extension named **DH_2**

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

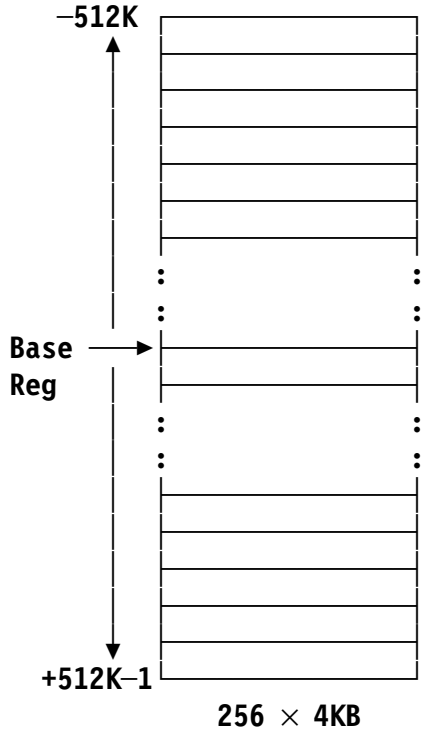


  - Displacement range $(-2^{19}, +2^{19}-1)$ rather than (0,4095)
- Address calculation adds base/index register contents as appropriate
  - Number of significant digits depends on current addressing mode
- If the DH field is zero, get usual 12-bit displacement

- 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

```
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 are underscored

- New instruction formats with 2-byte and 4-byte immediate operands
- 4-byte instruction:

opcode	R_1	op	RI_2

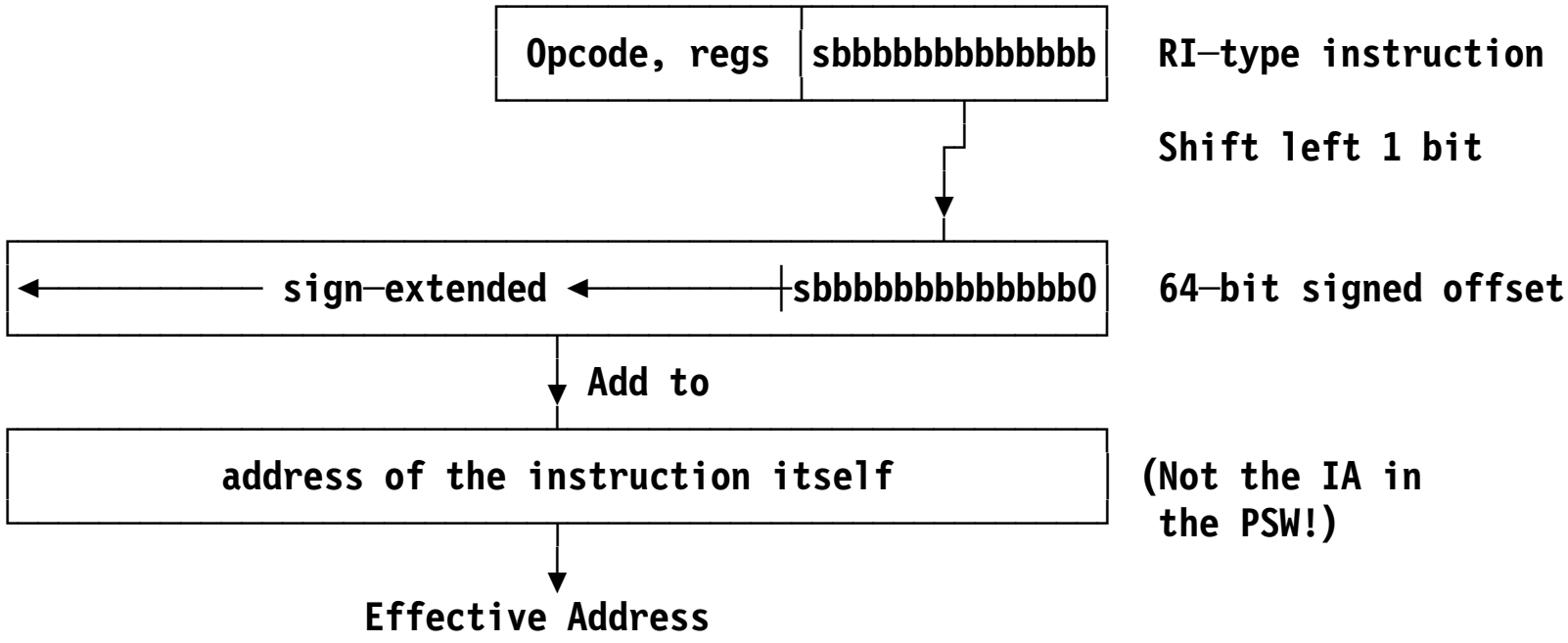
RI_2 range: $-2^{15} \leq I_2 \leq 2^{15}-1$, or $-32768 \leq I_2 \leq 32767$

- 6-byte instruction:

opcode	R_1	op	RI_2

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

- Address generation:



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

- 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”)

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”

- 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]

EXRL Execute Relative Long: no base register required

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

- Branch and save instructions

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

- 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

- 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

- Load or store action depends on Condition Code setting

Operation	Operand length	
	32 bits	64 bits
Load Register	LROC	LROCG
Load	LOC	LOCG
Store	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

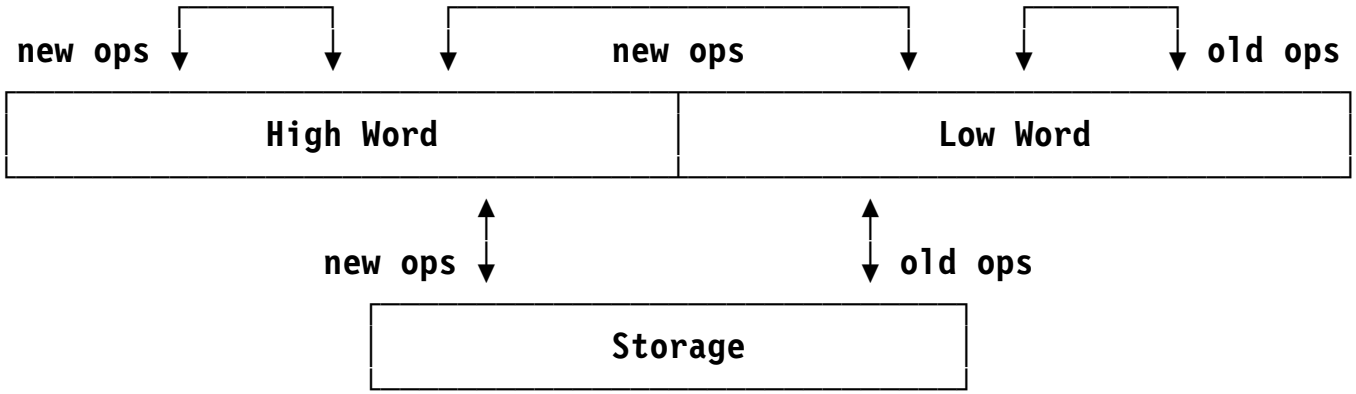
Old Way

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

New Way

```
     LR     2,0    Guess
     CR     0,1    Compare
     LROCL  2,1    Load if c(R0)<c(R1)
```

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



- 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

```
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
```

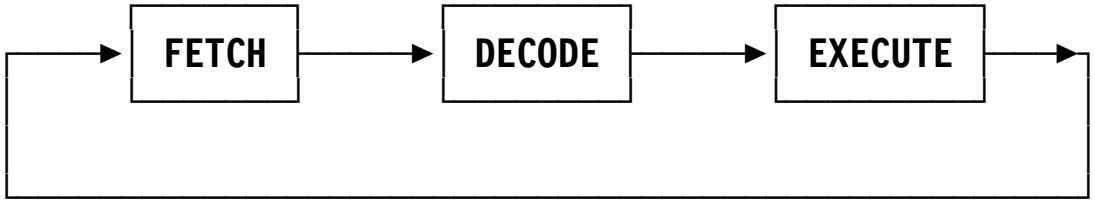
- These instructions let you preserve a value without first copying it:

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

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    0H
          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

Old Way

```
ST   2,Result
CLC  Result,OldValue

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

New Way

```
ST   2,Result
CL   2,OldValue

MVC  WorkArea(8),Data
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
```

- AHI vs. LA

```
L     0,0(,6)          Load GR0
LA    6,4(,6)          Increment pointer
```

  may be slightly faster than

```
L     0,0(,6)          Load GR0
AHI   6,4              Increment pointer
```

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

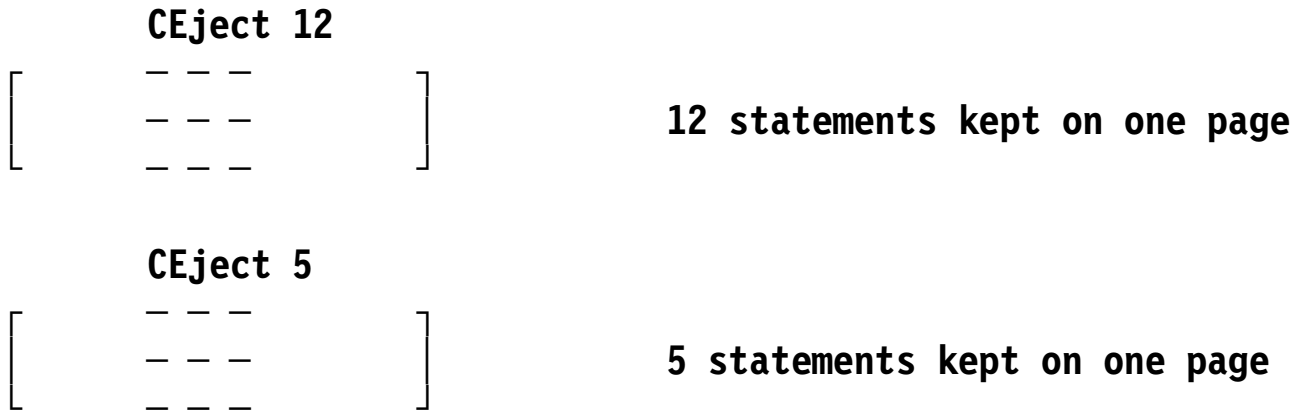
- Be very careful if you use LA, LAY for arithmetic: the results depend on the current addressing mode

- 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 counts lines remaining on the page, ejects if not enough
- Improved readability improves understanding

- 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...
         - - -
Code     LOCTR ,
         - - -
```

- Shows how you can keep related statements together in the source file

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

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!

- 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

- 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.

- "How Do You Do What You Do When You're a CPU? Two." SHARE Feb. 2005, Session 2835
- "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
- "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
- "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