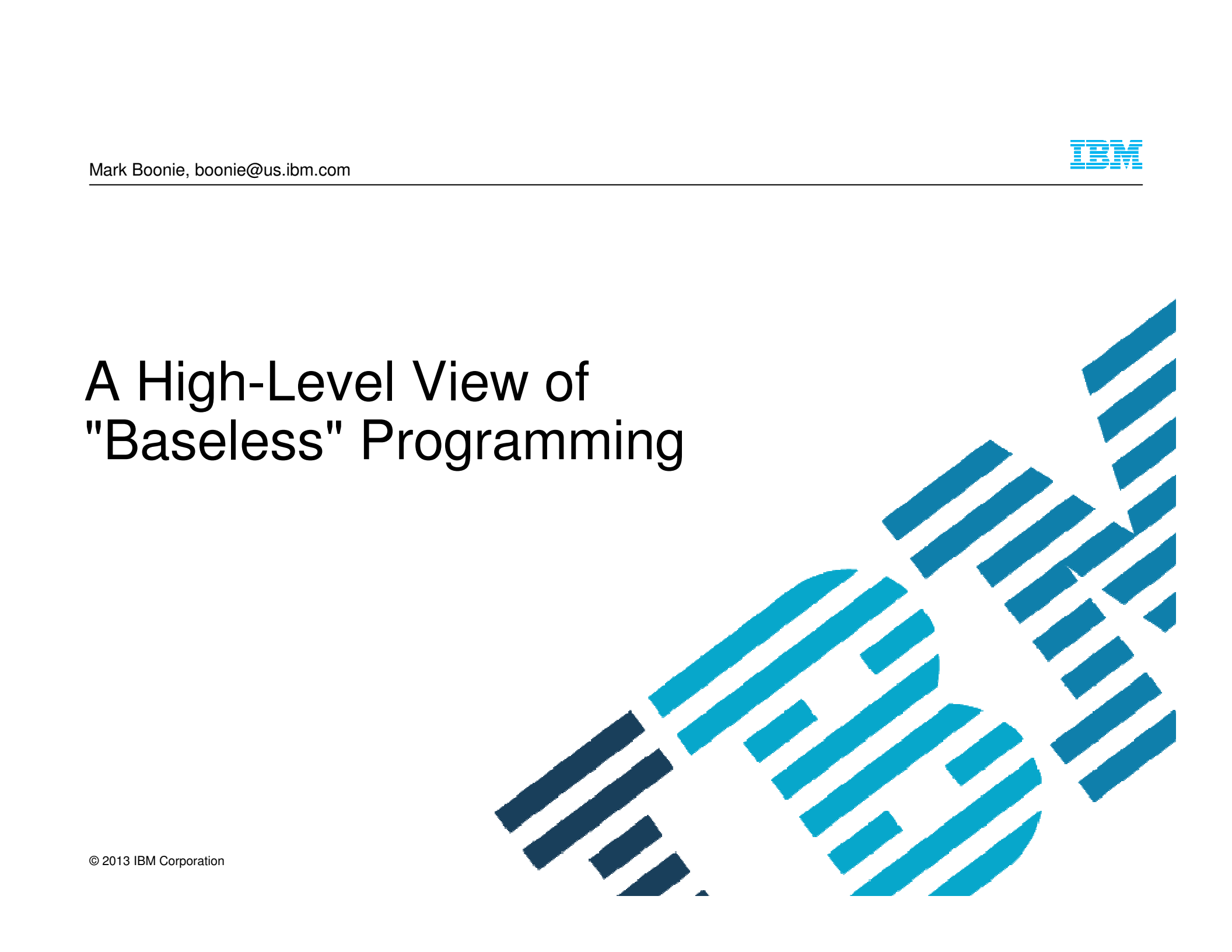
Mark Boonie, boonie@us.ibm.com



# A High-Level View of "Baseless" Programming

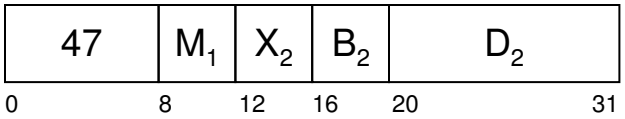
# Objective

- This course is for programmers that are familiar with z/Architecture and High-Level Assembler (HLASM) concepts.
- This is a technical course that primarily focuses on topics from a programmer point of view.
- General background in Computer Science is needed to understand the general computing concepts discussed in the material.

# History

- Most of the common Branch instructions are unchanged from S/360 (1964).
- BRANCH ON CONDITION

BC $M_1,D_2(X_2,B_2)$ [RX-b]




- Unconditional branch
  - Special case of BRANCH ON CONDITION
  - Mask is B'1111' (always branch)
- BRANCH ON COUNT (BCT)
- BRANCH ON INDEX HIGH (BXH), BRANCH ON INDEX LOW OR EQUAL (BXLE)

# History (cont.)

- Some instructions have had similar replacements.
  - BRANCH AND LINK (BAL) has been mostly replaced by BRANCH AND SAVE (BAS).

# Branching

- A branch instruction (in any of the previous flavors) passes control from one location in the code to a new location.

- In order to reach the new location, a base register must be defined to "cover" the location being branched to.

```
00012960                                57520=CCIOFLIH DS    0D

                      00012960          57526=         USING CCIOFLIH,R9

000129F2 B235 0200    00000200          57596=         TSCH  PFXIRB
000129F6 4770 913E              00012A9E  57597=         BNZ   FLIH0020
000129FA 9180 2001    00000001 00000080  57598=         TM    LDEVMNT,L'LDEVMNT
000129FE 47E0 9172              00012AD2  57599=         BNO   FLIH0040
```

# Branching (cont.)

- The format of the "target" has always been D(X,B).

- The displacement is 3 hex digits, meaning the target must be less than 4096 bytes from the address in the base register.
  - Half of all displacements can't be used because they're odd.

```
00012960                                57520=CCIOFLIH DS    0D

                      00012960          57526=         USING CCIOFLIH,R9

000129F2 B235 0200    00000200          57596=         TSCH  PFXIRB
000129F6 4770 913E              00012A9E  57597=         BNZ   FLIH0020
000129FA 9180 2001    00000001 00000080  57598=         TM    LDEVMNT,L'LDEVMNT
000129FE 47E0 9172              00012AD2  57599=         BNO   FLIH0040
```

# Larger Programs

- 4096 bytes used to be a large program; not so any more.
- Larger programs require multiple base registers.
- Example from CCCPSE:

```
USING CCDINIT,R7,R8,R4,R5      CPSE CODE BASE REGS
```

- This allows the program great flexibility in calling various routines, but at the expense of 4 registers.
  - 4 registers = 25% of all general registers

# Handling Larger Programs

- There is no one "correct" way to handle addressability in a large program.
  - Each subroutine may have its own base register.
    - This approach requires repeated saving and restoring of caller's base registers.
  - All subroutines may use the same base registers.
    - Saving and restoring of caller's base registers is not required.
    - Fewer registers are available to address other structures or for calculations.
- Merging code from two programs with different approaches is difficult.

# Program Growth

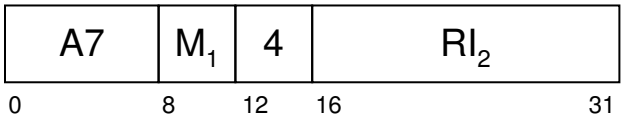
- As the program grows, the number of available registers shrinks.
- Reduced register availability leads to less efficient code.
  - Data must be kept in storage instead of registers.
    - Access to storage is slower than access to registers.
  - Registers are often saved and restored in order to free them for other uses.
- Eventually, additional growth becomes difficult.
  - Segmenting and reorganizing programs wastes valuable programmer resources.
  - An addressability shortage usually comes as an "Oh No!" surprise at the worst possible moment, sometimes adding hours or days to an otherwise simple project.
- Another example from CCCPSE:

```
USING CCDINIT,R7,R8        CPSE CODE BASE
...
USING CCDINIT,R7,R8,R4     USE 3 BASE REGISTERS
...
USING CCDINIT,R7,R8,R4,R5 CPSE CODE BASE REGS
```

# Branch Relative Instructions

- Branch Relative instructions address the addressability and growth problems
- BRANCH RELATIVE ON CONDITION

BRC $M_1$,$RI_2$ [RI-c]




- Branch Relative Unconditional (special case of BRC) - BRU
- BRCT, BRXH, BRXLE
- There are also instruction flavors that work with 64-bit registers.
  - BRCTG, BRXHG, BRXLG (not BRXLEG)
- To convert a based branch to a relative branch, change the 'B' to a 'BR' or to a 'J'
  - Exception: B → BRU (to avoid conflict with BR, which is the RR format of Branch)
  - Branch Relative instructions are usually called Jump instructions.

# Branch Relative Instructions (cont.)

- Instead of a base/displacement, Branch Relative instructions specify a target as a signed number of halfwords (not bytes) away from the instruction itself
- Based branching:

```
00012960                                57520=CCIOFLIH DS    0D

                      00012960          57526=         USING CCIOFLIH,R9

000129F2 B235 0200    00000200          57596=         TSCH  PFXIRB
000129F6 4770 913E              00012A9E  57597=         BNZ   FLIH0020
000129FA 9180 2001    00000001  00000080  57598=         TM    LDEVMNT,L'LDEVMNT
000129FE 47E0 9172              00012AD2  57599=         BNO   FLIH0040
```

- Relative branching:

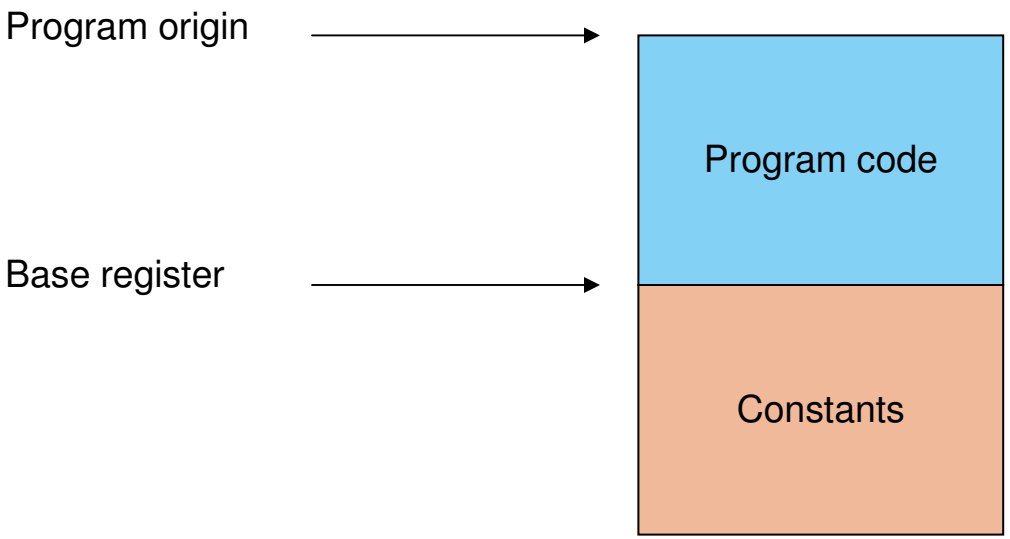
```
00012960                                57520=CCIOFLIH DS    0D

000129F2 B235 0200    00000200          57596=         TSCH  PFXIRB
000129F6 A774 0054              00012A9E  57597=         JNZ   FLIH0020
000129FA 9180 2001    00000001  00000080  57598=         TM    LDEVMNT,L'LDEVMNT
000129FE A7E4 006A              00012AD2  57599=         JNO   FLIH0040
```

# Branch Relative Instructions (cont.)

- Since the number of halfwords is 16 bits, the target can be up to 64K-bytes away from the current instruction in either direction.
- The range of a Branch Relative instruction is identical to the range of 16 4K-byte registers.
- While a base register may no longer be needed to branch from instruction to instruction, it may still be needed to address data (constants, work areas, save areas, etc.).
  - Some programmers use the term "baseless" to refer to executable code that uses no base register, with the assumption that data areas still require a base register.
- By placing the address of the data areas in the base register, a program may still greatly reduce the number of required base registers.

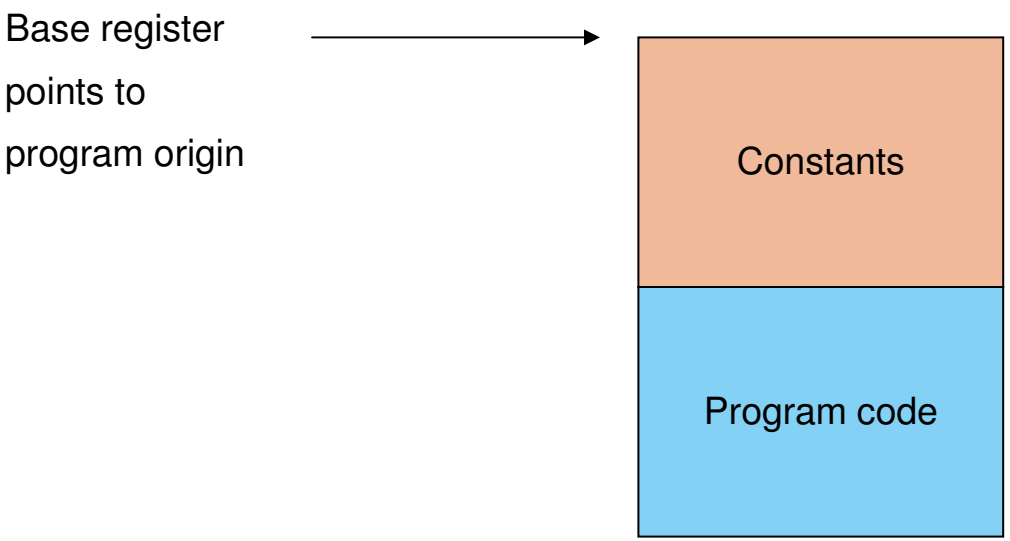
# Base Register Address

- Base register pointing to beginning of program is convenient, especially if the program is aligned on a cache line or page boundary.
- Base register pointing to middle of program is less convenient, making post-mortem analysis more difficult.



# Putting Constants First

- Base register points to beginning of program.
- Constants, work areas, etc., reside at the beginning and are pointed to by the base register.
- Code follows constants.



# Organizing Data Areas

- For data areas that are used by only a single routine, it is desirable to have the definition of the area visually close to the code that uses it.

```
RTN1     DC    0H                      Start of routine
...
         STNSM MASKSAVE,B'11111100' Disable interruptions
...
MASKSAVE DS    X                       System mask save area

*        End of RTN1
```

- This is difficult if the code is not covered by a base register.

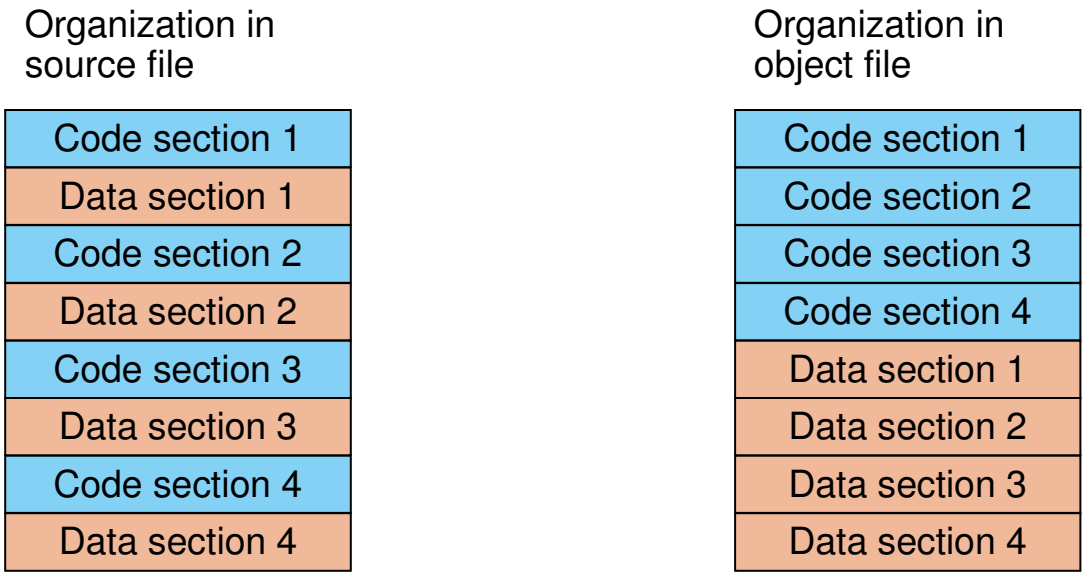
# Keeping Data Areas “Near” the Code (cont.)

- Define a new location counter and allow HLASM to keep the data organized.

```
RTNDATA  LOCTR ,                        Location counter for data
RTNCODE  LOCTR ,                        Location counter for code
...
*        Start of RTN1

RTNCODE  LOCTR ,                        Switch to the code LOCTR
RTN1     DC    0H                       Start of routine
...
         STNSM MASKSAVE,B'11111100' Disable interruptions
...
RTNDATA  LOCTR ,                        Switch to the data LOCTR
MASKSAVE DS    X                        System mask save area
RTNCODE  LOCTR ,                        Switch back to the code LOCTR

*        End of RTN1
```

# Keeping Data Areas “Near” the Code (cont.)

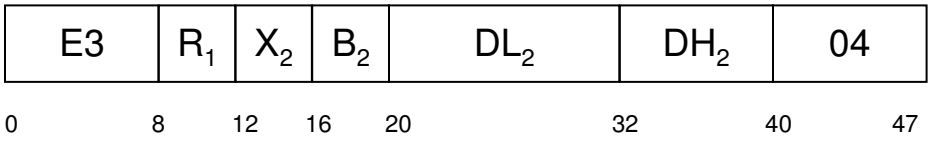
- The same location counters can be used repeatedly by multiple routines.



# Additional Instructions - Long Displacements

- To allow more flexibility in accessing large data areas, many instructions have been modified to use long displacements.
- Historically, displacements have been unsigned 12-bit integers, allowing for a displacement up to 4095 bytes from the base register.
- Long displacements are signed 20-bit integers, allowing for a displacement up to 512K-bytes from the base register.

LG $R_1,D_2(X_2,B_2)$ [RXY-a]

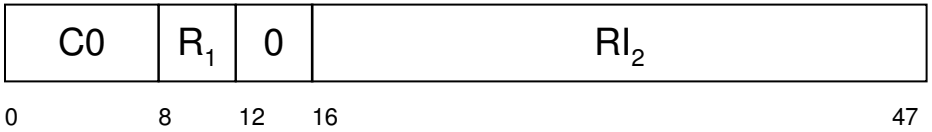



- Most instructions that work with 64-bit storage operands use long displacement.

# Additional Instructions – Relative-Long

- A small set of instruction use long versions of relative operands.

LARL R1,RI2 [RIL-b]




- Long relative operands specify an operand as a signed 32-bit integer representing a number of halfwords that are added to the address of the instruction.
  - This allows the operand to be up to 4G-bytes away from the instruction.
- Long relative instructions initially included LARL, BRCL, and BRASL.
- Later additions included:
  - Load, Load Logical, Load Halfword, and Load Logical Halfword
  - Store and Store Halfword
  - Compare, Compare Logical, and Compare Halfword
  - Execute

# Writing Baseless Code in z/TPF

- For E-type programs, the BEGIN macro can specify that no base register will be used.

```
BEGIN NAME=CYMA,                                    +
      IBM=YES,                                      +
      BASELESS=YES,                                 +
      BASE=NONE,                                    +
      TV=CYMB
```

- BASE=NONE indicates that no USING is to be generated for R8.
  - This parameter is not unique to baseless code; it could be used if the programmer intends to establish addressability by hand.
- BASELESS=YES indicates that other macros invoked by this program are to generate baseless code.
  - Macros will not assume that code is covered by a base register.
  - May still be specified with BASE=R8.
- A few other macros allow a BASE=NONE parameter to indicate that there is no active base register.

# Writing Baseless Code in z/TPF (cont.)

- For CP programs, the BEGIN macro is not used.
  - Programmers typically establish addressability by hand.
- The CLNKC macro is used to invoke subroutines:

```
CLNKC RTN=FINDNDS,     Go set up displacement descriptors +
      BASE=NONE,       for this area                      +
      LINK=R14,                                           +
      TYPE=EXT
```

- BASE=NONE indicates that linkage is to be performed using BRASL.
- The TYPE= parameter can be any value except POINTER
  - TYPE=POINTER requires that a pointer to the routine be loaded into a register, which then becomes the base register of the invoked routine.
- Values for BASE= parameter for RLNKC and SLNKC should be identical to the corresponding CLNKC.

# Writing Baseless Code in z/TPF (cont.)

- Most macros generate baseless code.

- A small number of macros generate different code depending on the setting of the BASELESS= parameter on the BEGIN macro.
  - This was done to avoid changing expansion sizes for programs that were reassembled but not modified.

- There may be some infrequently-used macros that do not assemble correctly without an active base register. If a user would like to have such a macro updated by IBM, please submit a requirement.

# Acknowledgements


- Thanks and gratitude are owed to the following individuals whose work I have borrowed for portions of this presentation:
  - John Ehrman, IBM
  - Ed Jaffe, Phoenix Software International, Inc.