

SHARE

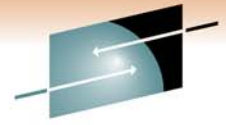
Technology • Connections • Results

Reducing Base Register Utilization: How to “Jumpify” Your Programs

Edward E. Jaffe
Phoenix Software International, Inc.

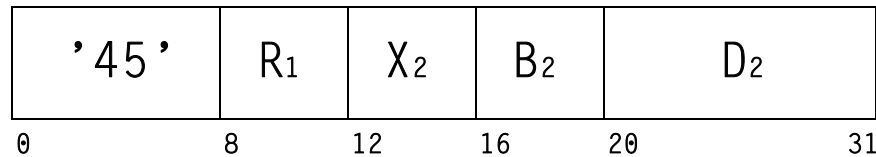
Monday February 28, 2011
Session 8548



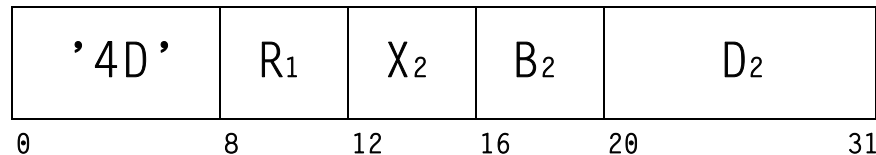


ESA/390 Based Branch Instructions

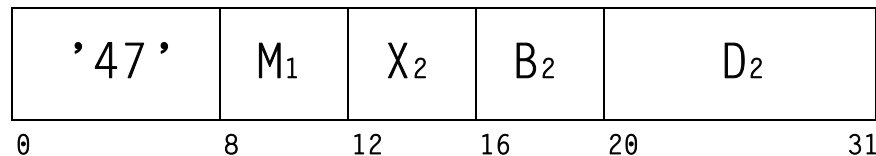
BAL $R_1, D_2(X_2, B_2)$ [RX]



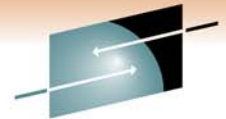
BAS $R_1, D_2(X_2, B_2)$ [RX]



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

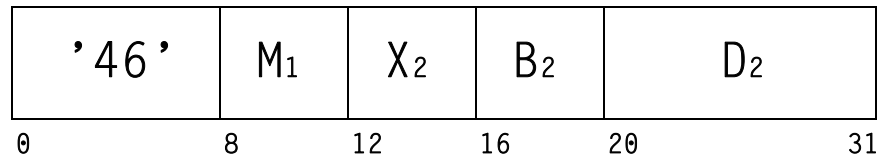


- RX and RS format.
- Displacement odd or even -- always positive.
 - Only ½ of displacements result in a valid branch.
 - 12-bit displacement means max is 4095 bytes (x'FFF').
 - Odd branch address results in PIC 0006.
- Used for conditional program logic and “near” subroutine calls.

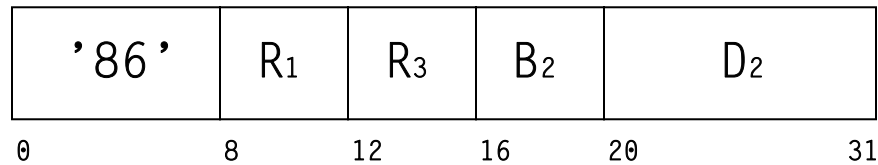


ESA/390 Based Branch Instructions

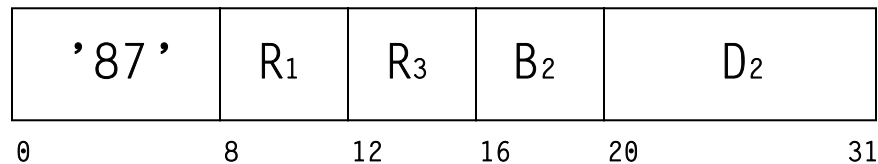
BCT $R_1, D_2(X_2, B_2)$ [RX]



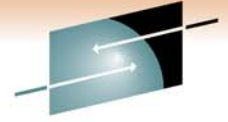
BXH $R_1, R_3, D_2(X_2, B_2)$ [RS]



BXLE $R_1, R_3, D_2(X_2, B_2)$ [RS]

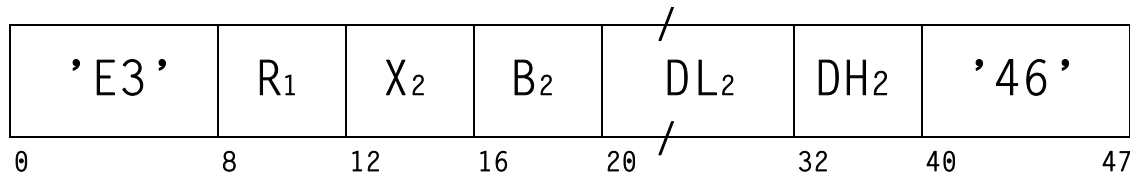


Based Branch Instructions Added with z/Architecture

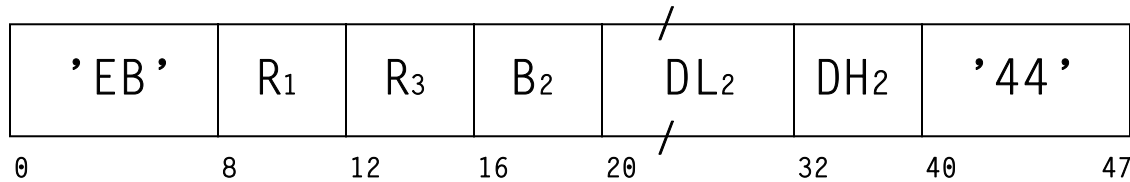


SHARE
Technology • Connections • Results

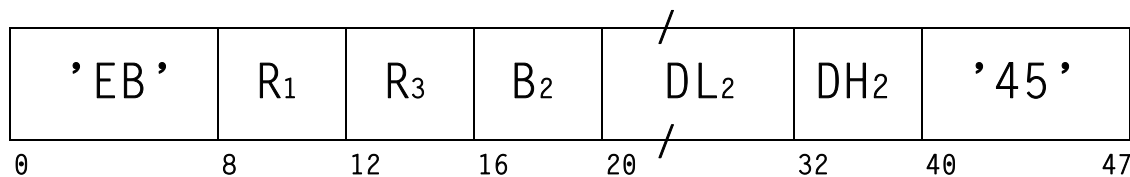
BCTG $R_1, D_2(X_2, B_2)$ [RXY]

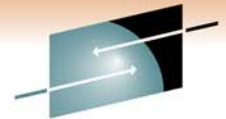


BXHG $R_1, R_3, D_2(X_2, B_2)$ [RSY]



BXLEG $R_1, R_3, D_2(X_2, B_2)$ [RSY]



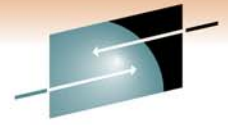


BC Extended Mnemonics

BH	Branch on High	BC 2
BL	Branch on Low	BC 4
BE	Branch on Equal	BC 8
BNH	Branch on Not High	BC 13
BNL	Branch on Not Low	BC 11
BNE	Branch on Not Equal	BC 7

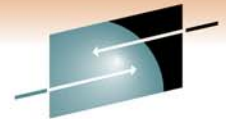
BO	Branch if Ones	BC 1
BM	Branch if Mixed	BC 4
BZ	Branch if Zero	BC 8
BNO	Branch if Not Ones	BC 14
BNM	Branch if Not Mixed	BC 11
BNZ	Branch if Not Zero	BC 7

BP	Branch on Plus	BC 2
BM	Branch on Minus	BC 4
BO	Branch on Overflow	BC 1
BZ	Branch on Zero	BC 8
BNP	Branch on Not Plus	BC 13
BNM	Branch on Not Minus	BC 11
BNZ	Branch on Not Zero	BC 7
BNO	Branch on Not Overflow	BC 14



About (Based) Branches

- For decades, all branches were based. There was no need for differentiation.
- The use of (based) branch instructions requires that nearly every line of program code be "covered" by a base register.
- Based branches are subject to processor pipeline delays due to Address Generation Interlock (AGI).
- Base registers are loaded more often than programmers realize (e.g., reloading registers on return from subroutine).
- For historical reasons, many experienced assembler language programmers continue to use the word "branch", in an unqualified manner, to mean a based branch.



Sample (Based) Branch Usage

```
. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.
.
.
.0000000C 9180 C084          00000084      28              TM      FLAG,BIT
.00000010 4780 C01C          0000001C      29              BC      8,NOTSET
.00000014 4DE0 C084          00000084      30              BAS     R14,ITS_ON
.00000018 47F0 C020          00000020      31              BC      15,CONTINUE
.0000001C
.0000001C 4DE0 C084          00000084      33              BAS     R14,ITS_OFF
.00000020      34 CONTINUE DC      0H
```

```
. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.
.
.
.0000000C 9180 C084          00000084      28              TM      FLAG,BIT
.00000010 4780 C01C          0000001C      29              BZ      NOTSET
.00000014 4DE0 C084          00000084      30              BAS     R14,ITS_ON
.00000018 47F0 C020          00000020      31              B       CONTINUE
.0000001C
.0000001C 4DE0 C084          00000084      33              BAS     R14,ITS_OFF
.00000020      34 CONTINUE DC      0H
```

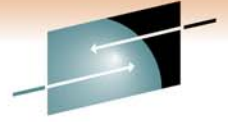
Sample Program >4KiB with Complete Base Register Coverage

- As the program grows, the number of available registers shrinks.
- Reduced register availability leads to less efficient code.
- Eventually, additional growth becomes impossible.

```

. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.00000000                                2 MULTIBAS CSECT ,
.                                3 *          [save registers]
.00000000 18CF                                4          LR      R12,R15
.00000002 41B0 C800                                5          LA      R11,2048(,R12)
.00000006 41B0 B800                                6          LA      R11,2048(,R11)
.0000000A 41A0 B800                                7          LA      R10,2048(,R11)
.0000000E 41A0 A800                                8          LA      R10,2048(,R10)
.00000012 4190 A800                                9          LA      R9,2048(,R10)
.00000016 4190 9800                               10         LA      R9,2048(,R9)
.                                11         USING MULTIBAS,R12,R11,R10,R9
.                                15 *          .
.                                16 *          . (16KiB code & constants)
.                                17 *          .
.00000030                                18         LTORG ,
.                                41         END ,

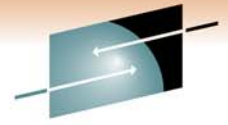
```

SHARE
Technology • Connections • Results

Mitigating Limitations on Addressability

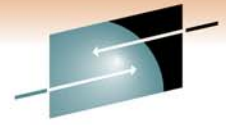
- Over the years, many smart programmers invented clever techniques to mitigate limitations on addressability.
- Too many to list. (I probably would not be able to imagine them all anyway.)
- Probably no single "best" solution.
- I'll show one example for illustrative purposes.



Mitigating Limitations on Addressability

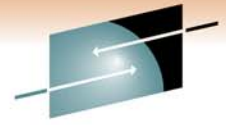
- Using three base registers, this program can support many 4KiB routines.
- New base register needed only if mainline routine requires expansion. (Each subroutine requires 12 mainline bytes.)

Loc	Object Code	Addr1	Addr2	Stmt	Source Statement
.00000000		00000000	00003EE0	2	MULTIRTN CSECT ,
.				3	* [save registers]
.00000000	18CF			4	LR R12,R15
.00000002	41B0 C800		00000800	5	LA R11,2048(,R12)
.00000006	41B0 B800		00000800	6	LA R11,2048(,R11)
.	R:CB	00000000		7	USING (MULTIRTN,MULTIMLX),R12,R11
.				8	* .
.0000000A	58F0 BF64		00001F64	9	L R15,RTN1A
.0000000E	0DEF			10	BASR R14,R15
.				11	* .
.00000010	58F0 BF68		00001F68	12	L R15,RTNnA
.00000014	0DEF			13	BASR R14,R15
.				18	* .
.				19	* . (8KiB mainline & constants)
.				20	* .
.00001F64	00001F82			21	RTN1A DC A(RTN1)
.00001F68	00002F30			23	RTNnA DC A(RTNn)
.				24	* .
.00001F70				25	LTORG ,
.		00001F81		29	MULTIMLX EQU *



Mitigating Limitations on Addressability

.		30	PUSH USING
.00001F81 00			
.00001F82		31 RTN1	DC OH
.		32 *	[save registers]
.00001F82 18AF		33	LR R10,R15
.	R:A 00001F82	34	USING (RTN1,RTN1X),R10
.		36 *	.
.		37 *	. (4KiB subroutine & constants)
.		38 *	.
.00002F30		39	LTORG ,
.	00002F30	40 RTN1X	EQU *
.		41	POP USING
.		42	PUSH USING
.00002F30		43 RTNn	DC OH
.		44 *	[save registers]
.00002F30 18AF		45	LR R10,R15
.	R:A 00002F30	46	USING (RTNn,RTNnX),R10
.		48 *	.
.		49 *	. (4KiB subroutine & constants)
.		50 *	.
.00003EE0		51	LTORG ,
.	00003EE0	52 RTNnX	EQU *
.		53	POP USING



A Clever Compiler-Only Solution

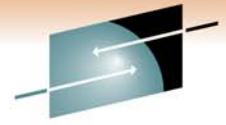
- Load constant values of 4KiB, 8KiB, and 12KiB into three index registers and load base registers 16KiB apart.
- Knowing the branch target, the compiler selects the appropriate base/index for each generated branch.
- Using this technique, five registers can cover 32KiB, six registers can cover 48KiB, etc.

0000-0FFF	Covered by 1 st base	BC xx,ddd(0,B1)
1000-1FFF	Covered by 1 st base + 4096	BC xx,ddd(I1,B1)
2000-2FFF	Covered by 1 st base + 8192	BC xx,ddd(I2,B1)
3000-3FFF	Covered by 1 st base + 12288	BC xx,ddd(I3,B1)
4000-4FFF	Covered by 2 nd base	BC xx,ddd(0,B2)
5000-5FFF	Covered by 2 nd base + 4096	BC xx,ddd(I1,B2)
6000-6FFF	Covered by 2 nd base + 8192	BC xx,ddd(I2,B2)
7000-7FFF	Covered by 2 nd base + 12288	BC xx,ddd(I3,B2)

Why Has the Industry Transitioned to Relative Branch?

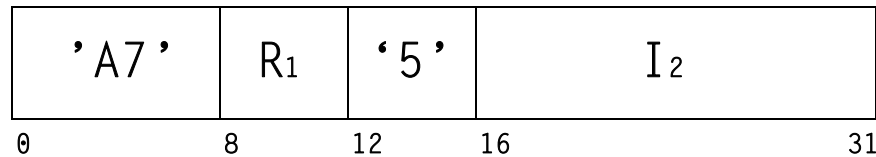


- Fetching of instructions by hardware does not depend upon base register contents. Rather, only the contents of the right half of the Program Status Word are needed.
- As you've seen 4KiB branch displacements are highly restrictive. Segmenting and reorganizing growing programs is a waste of precious manpower.
 - 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.
- Relative branch is better performing: not subject to AGI and uses a less complex address resolution scheme!
- Arguably one of the most important and useful improvements on the platform.

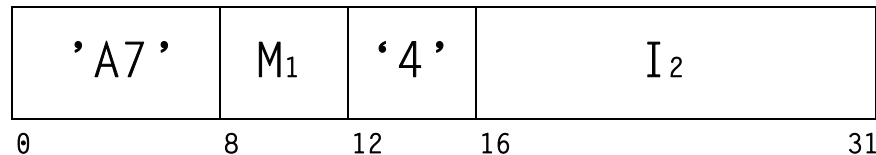


ESA/390 Relative Branch Instructions

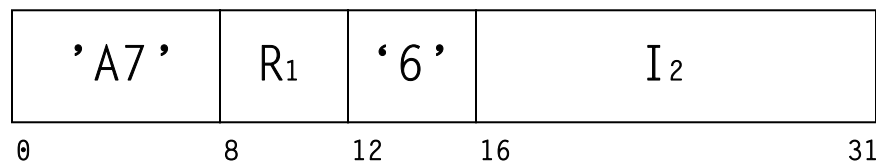
BRAS R₁, I₂ [RI]



BRC M₁, I₂ [RI]



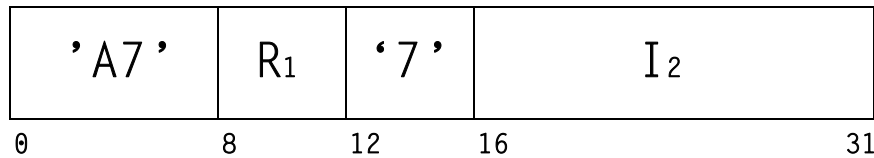
BRCT R₁, I₂ [RI]



- RI and RSI format.
- Offset from current instruction address is signed number of halfwords, represented by the immediate value.
 - Maximum valid offset is nearly $\pm 64\text{KiB}$.
- Originated on non-IBM PCMs in Japan.
- Implemented in IBM ESA/390 hardware as part of Relative-Immediate facility.

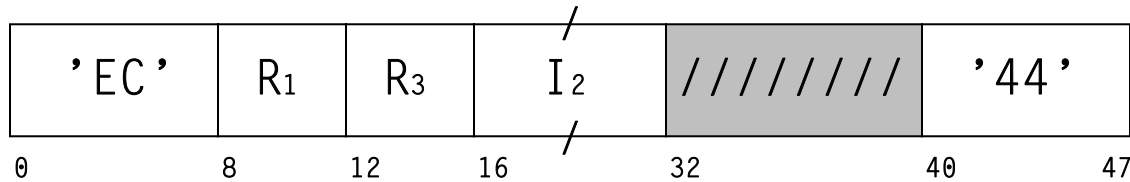
Relative Branch Instructions Added with z/Architecture

BRCTG R₁, I₂ [RI]

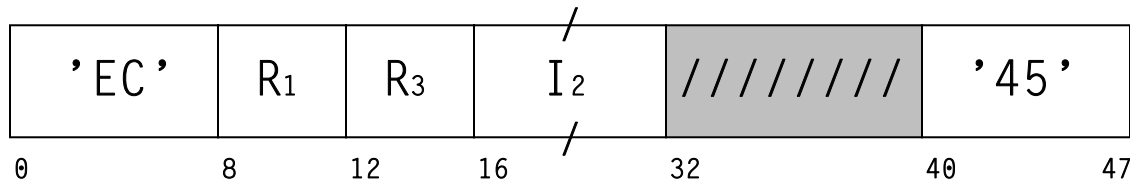


- RI and RIE format.

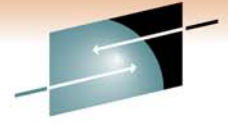
BRXHG R₁, R₃, I₂ [RIE]



BRXLG R₁, R₃, I₂ [RIE]

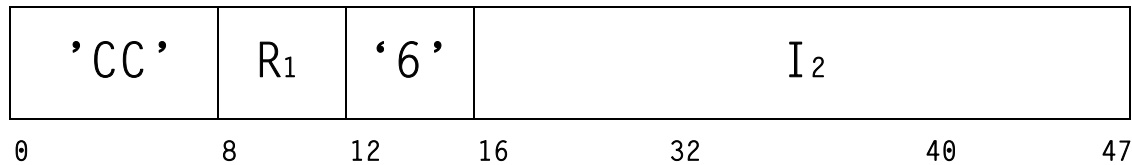


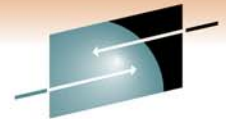
Relative Branch Instruction Added with zEnterprise



SHARE
Technology • Connections • Results

BRCTH R₁, I₂ [RIL-b]





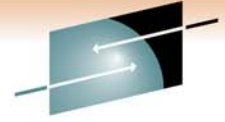
BRC/BRAS Extended Mnemonics

JH	Jump on High	BRC 2
JL	Jump on Low	BRC 4
JE	Jump on Equal	BRC 8
JNH	Jump on Not High	BRC 13
JNL	Jump on Not Low	BRC 11
JNE	Jump on Not Equal	BRC 7

JO	Jump if Ones	BRC 1
JM	Jump if Mixed	BRC 4
JZ	Jump if Zero	BRC 8
JNO	Jump if Not Ones	BRC 14
JNM	Jump if Not Mixed	BRC 11
JNZ	Jump if Not Zero	BRC 7

JP	Jump on Plus	BRC 2
JM	Jump on Minus	BRC 4
JO	Jump on Overflow	BRC 1
JZ	Jump on Zero	BRC 8
JNP	Jump on Not Plus	BRC 13
JNM	Jump on Not Minus	BRC 11
JNZ	Jump on Not Zero	BRC 7
JNO	Jump on Not Overflow	BRC 14

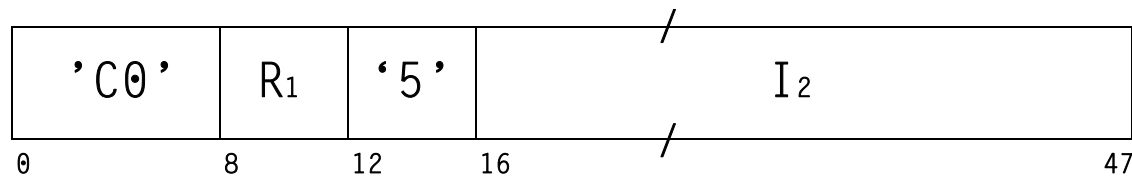
JAS	Jump and Save	BRAS
-----	---------------	------



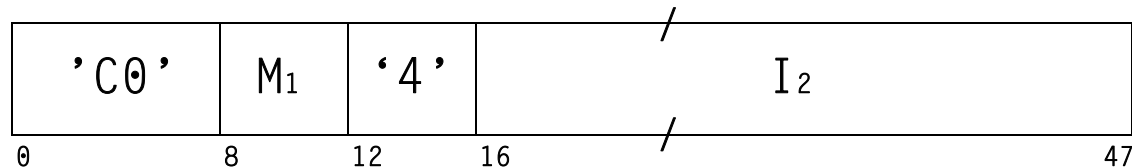
Relative Branch Long Instructions

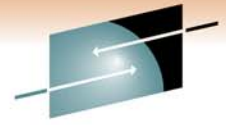
- Maximum valid offset is $\pm 4\text{GiB}$
- Available on machines that implement z/Architecture.
- So-called "N3" addition to ESA/390 instruction set.

BRASL R_1, I_2 [RIL]



BRCL M_1, I_2 [RIL]





BRCL/BRASL Extended Mnemonics

JLH	Jump on High	BRCL 2
JLL	Jump on Low	BRCL 4
JLE	Jump on Equal	BRCL 8
JLNH	Jump on Not High	BRCL 13
JLNL	Jump on Not Low	BRCL 11
JLNE	Jump on Not Equal	BRCL 7

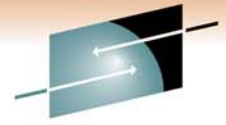
JLO	Jump if Ones	BRCL 1
JLM	Jump if Mixed	BRCL 4
JLZ	Jump if Zero	BRCL 8
JLNO	Jump if Not Ones	BRCL 14
JLNM	Jump if Not Mixed	BRCL 11
JLNZ	Jump if Not Zero	BRCL 7

JLP	Jump on Plus	BRCL 2
JLM	Jump on Minus	BRCL 4
JLO	Jump on Overflow	BRCL 1
JLZ	Jump on Zero	BRCL 8
JLNP	Jump on Not Plus	BRCL 13
JLNM	Jump on Not Minus	BRCL 11
JLNZ	Jump on Not Zero	BRCL 7
JLNO	Jump on Not Overflow	BRCL 14

JLU	Jump Unconditional	BRCL 15
-----	--------------------	---------

JASL	Jump and Save	BRASL
------	---------------	-------

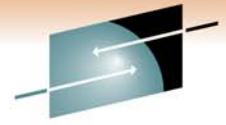
In each of the BRCL cases, **BR** can be substituted for **J**. But who cares?



Which Base Registers Are Eliminated?

- Program code no longer requires base register coverage.
- Some programmers like to use the term "baseless" to describe programs that don't use based branches.
- The latest hardware generations support relative data references for many new instructions. Nevertheless, programs are still expected to have base register coverage for constants (literals are a subset of constants).
- Non-reentrant programs are still expected to have base register coverage for local working storage. This might be the same base used for constants.

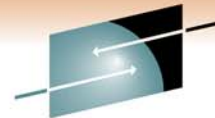
Note: From this point on, I will use the term "jump" to mean relative branch and the term "branch" to mean based branch.



Sample Jump Usage

```
. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.
.
.
.0000000C 9180 C084          00000084      29           TM      FLAG,BIT
.00000010 A784 0006          0000001C      30           BRC      8,NOTSET
.00000014 A7E5 0038          00000084      31           BRAS     R14,ITS_ON
.00000018 A7F4 0004          00000020      32           BRC      15,CONTINUE
.0000001C
.0000001C A7E5 0034          00000084      34           BRAS     R14,ITS_OFF
.00000020      35 CONTINUE DC      0H
```

```
. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.
.
.
.0000000C 9180 C084          00000084      28           TM      FLAG,BIT
.00000010 A784 0006          0000001C      29           JZ       NOTSET
.00000014 A7E5 0038          00000084      30           JAS      R14,ITS_ON
.00000018 A7F4 0004          00000020      31           J        CONTINUE
.0000001C
.0000001C A7E5 0034          00000084      33           JAS      R14,ITS_OFF
.00000020      34 CONTINUE DC      0H
```



Sample Jump Long Usage

```
. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.
.
.
.0000000C 9180 C08C          0000008C      28          TM      FLAG,BIT
.00000010 C084 0000 0009          00000022      29          BRCL  8,NOTSET
.00000016 C0E5 0000 003B          0000008C      30          BRASL R14,ITS_ON
.0000001C C0F4 0000 0006          00000028      31          BRCL  15,CONTINUE
.00000022          00000022      32 NOTSET  DC      0H
.00000022 C0E5 0000 0035          0000008C      33          BRASL R14,ITS_OFF
.00000028          00000028      34 CONTINUE DC      0H
```

```
. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.
.
.
.0000000C 9180 C08C          0000008C      29          TM      FLAG,BIT
.00000010 C084 0000 0009          00000022      30          JLZ   NOTSET
.00000016 C0E5 0000 003B          0000008C      31          JASL  R14,ITS_ON
.0000001C C0F4 0000 0006          00000028      32          JLU   CONTINUE
.00000022          00000022      33 NOTSET  DC      0H
.00000022 C0E5 0000 0035          0000008C      34          JASL  R14,ITS_OFF
.00000028          00000028      35 CONTINUE DC      0H
```

Sample Base Register Coverage For Constants Only

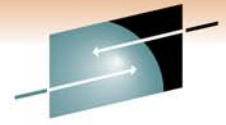
- One base register still covers 4KiB. If your program will use instructions that support long (20-bit) displacements, you can extend this.
- This example uses LR/AHI instead of LARL. If your program will run only on machines that implement z/Architecture, you can use LARL even in ESA/390 mode.

```

.  Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.00000000          00000000 00000091      2 NOCODBAS CSECT ,
.          3 *                [save registers]
.00000000 18CF          4                LR    R12,R15
.00000002 A7CA 0078          00000078      5                AHI   R12,CONST-NOCODBAS
.          R:C 00000078      6                USING CONST,R12
.          30 *                .
.          31 *                . (64KiB code)
.          32 *                .
.00000076 0000          33 CONST    DC    0D
.00000078          34 *                .
.          35 *                . (4KiB constants)
.          36 *                .
.00000080          37                LTORG ,
.          41                END    ,

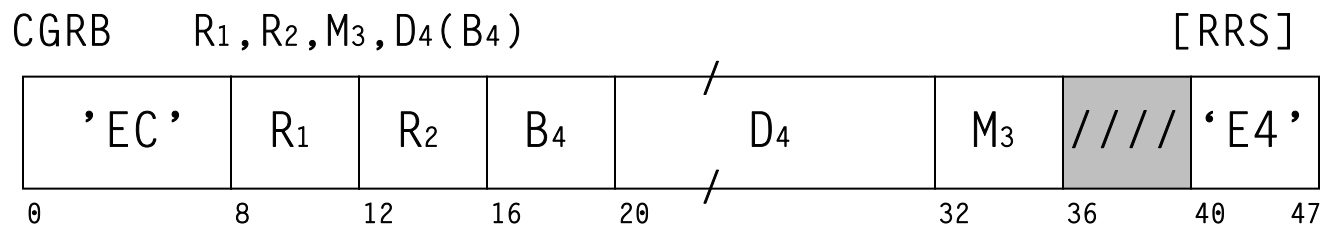
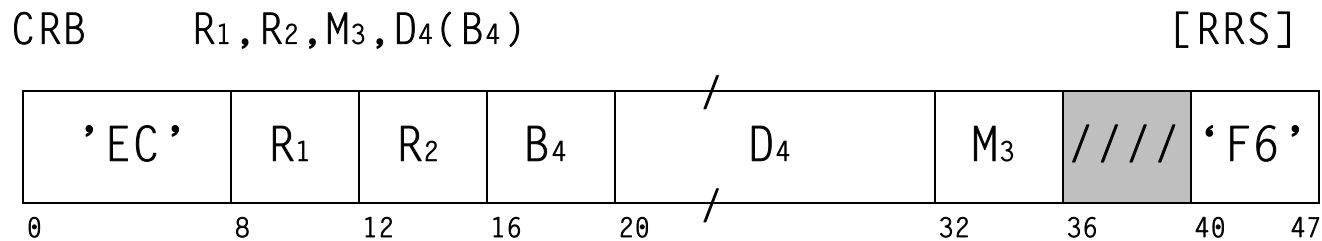
```

One AHI advances up to 32KiB only.



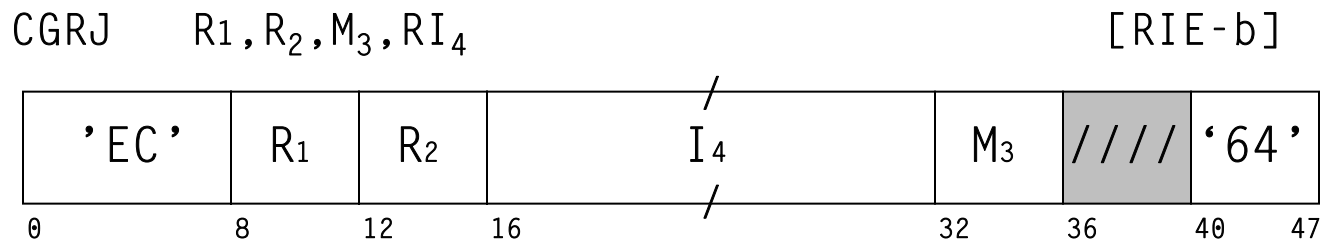
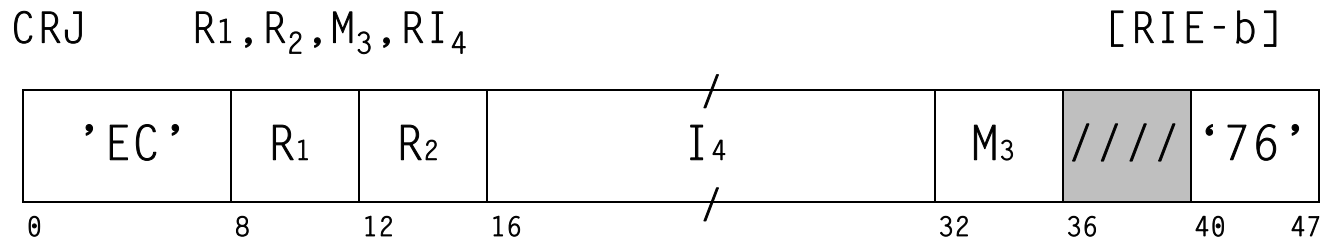
Compare and Branch Instructions

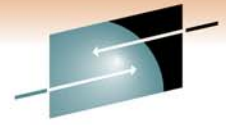
- Maximum branch displacement is +4KiB
- Available starting with IBM System z10 processors.



Compare and Branch Relative Instructions

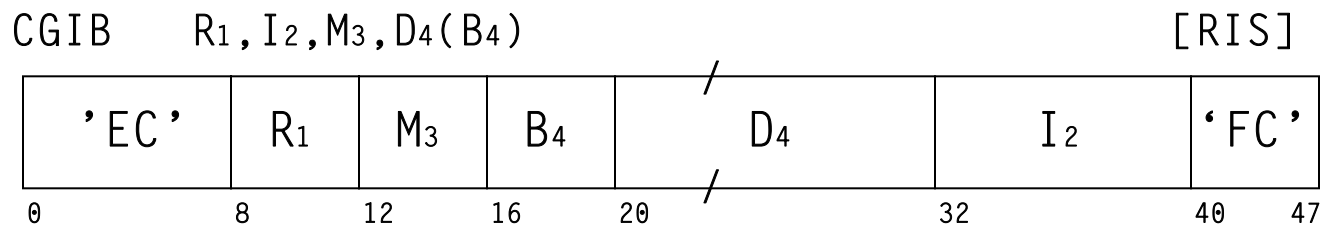
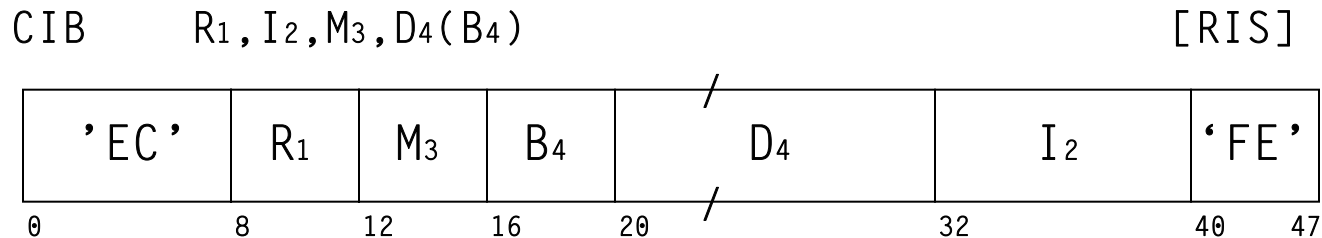
- Maximum valid offset is $\pm 64\text{KiB}$ (no “long” 32-bit form)
- Available starting with IBM System z10 processors.





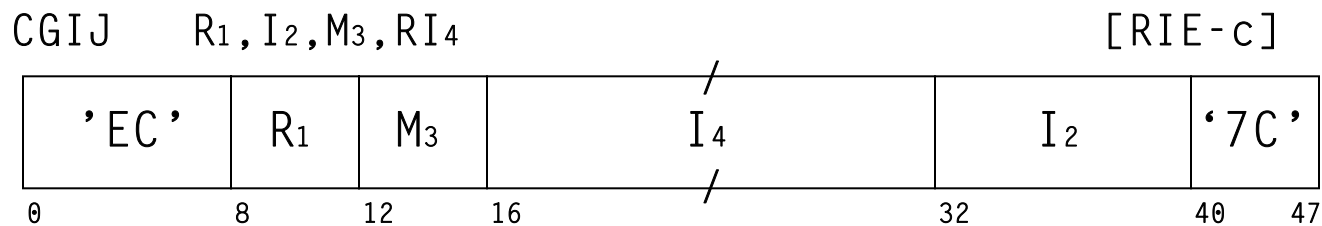
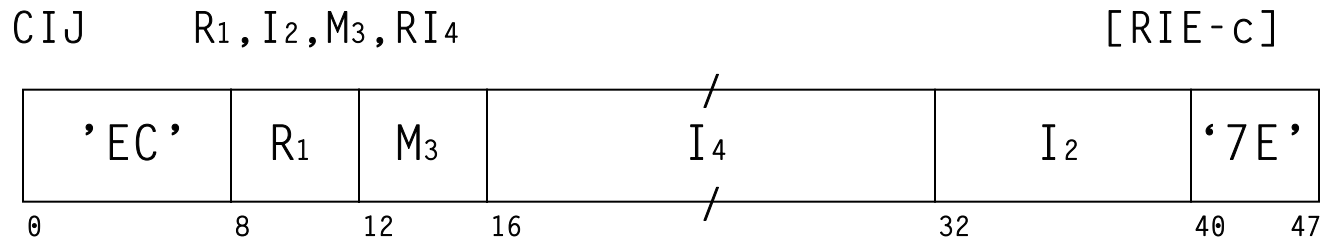
Compare Immediate and Branch Instructions

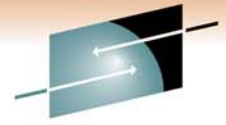
- Maximum branch displacement is +4KiB
- Available on System z10 and higher processors.



Compare Immediate and Branch Relative Instructions

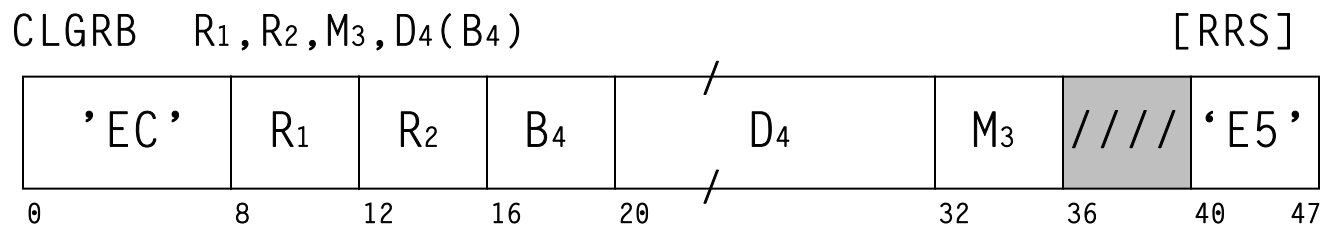
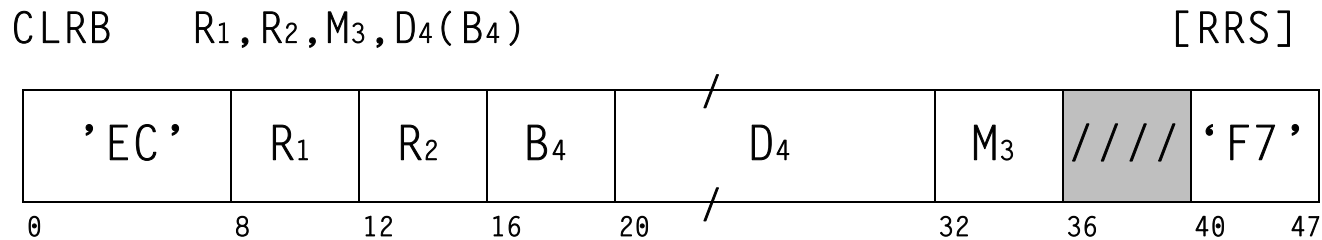
- Maximum valid offset is $\pm 64\text{KiB}$ (no “long” 32-bit form)
- Available on System z10 and higher processors.





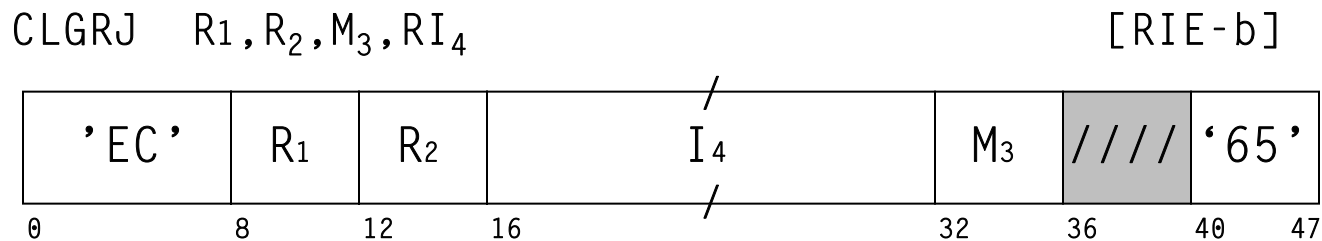
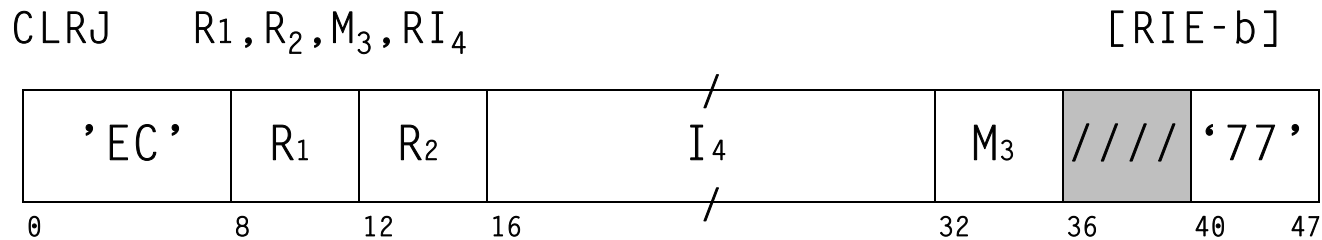
Compare Logical and Branch Instructions

- Maximum branch displacement is +4KiB
- Available starting with IBM System z10 processors.

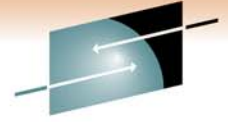


Compare Logical and Branch Relative Instructions

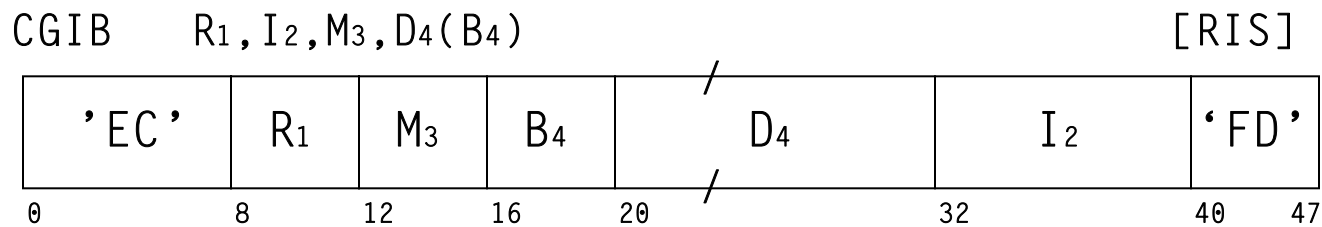
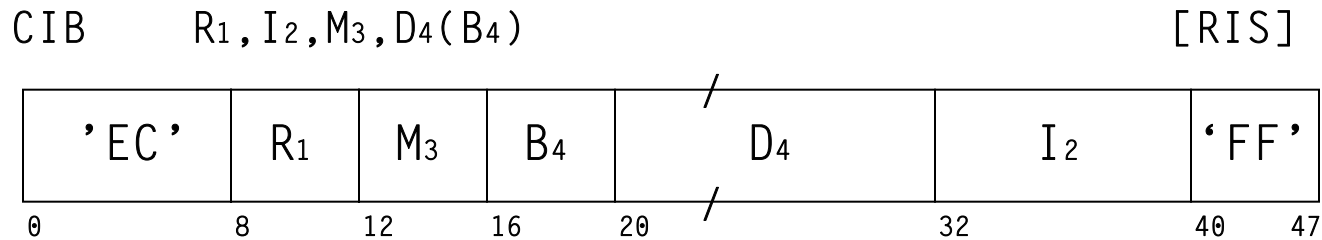
- Maximum valid offset is $\pm 64\text{KiB}$ (no “long” 32-bit form)
- Available starting with IBM System z10 processors.



Compare Logical Immediate and Branch Instructions

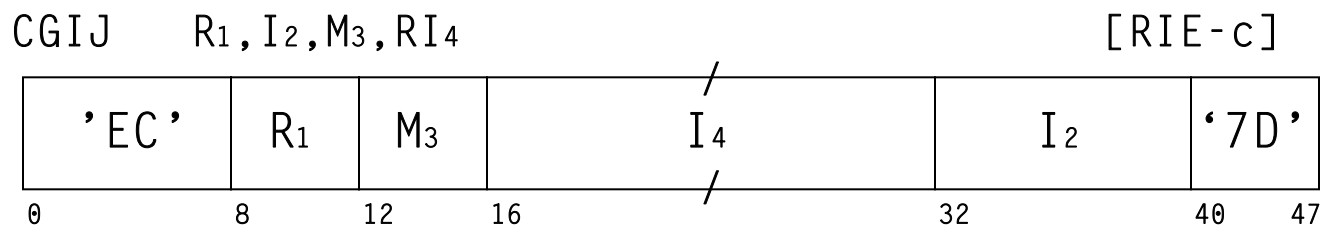
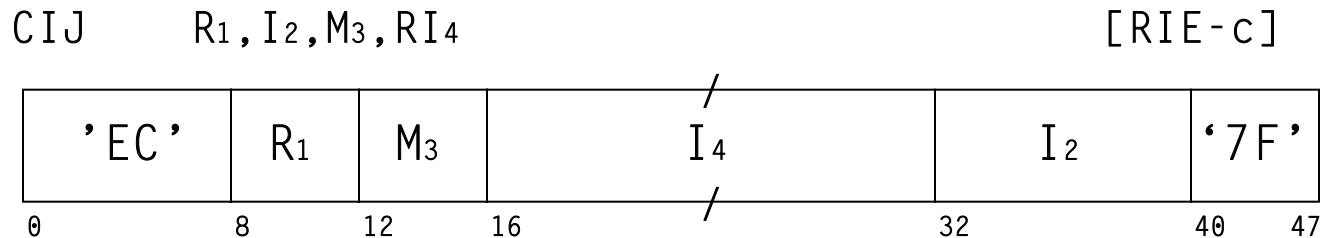


- Maximum branch displacement is +4KiB
- Available on System z10 and higher processors.



Compare Logical Immediate and Branch Relative Instructions

- Maximum valid offset is $\pm 64\text{KiB}$ (no “long” 32-bit form)
- Available on System z10 and higher processors.



Compare and Branch Extended Mnemonics

- Extended mnemonics for the compare and branch instructions follow familiar pattern.
- Rather than explicitly specifying M_3 value, you can append a suffix from the following table:

Suffix Chars	Meaning	Mask Field
E	Equal	8
H	First operand high	2
L	First operand low	4
NE	Not equal	6
NH	First operand not high	12
NL	First operand not low	10

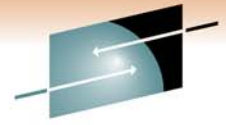
Example:

```
CGIJNE R10,-123,LABEL
```

is equivalent to:

```
CGIJ R10,-123,6,LABEL
```

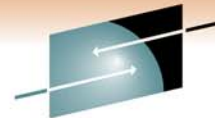
- No mnemonics for “zero”, “ones”, “positive”, “mixed”, etc.



Sample Compare and Branch Usage

```
.  Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.
.
.
.0000000C  ECAB  C01A  70F6      0000001A  31          CRB   R10,R11,7,NOTEQUAL
.00000012  4DE0  C022      00000022  32          BAS   R14,SAME
.00000016  47F0  C01E      0000001E  33          BC    15,CONTINUE
.0000001A
.0000001A  4DE0  C022      00000022  35          BAS   R14,DIFFERENT
.0000001E  36  CONTINUE DC    0H
```

```
.  Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.
.
.
.0000000C  ECAB  C01A  70F6      0000001A  31          CRBNE R10,R11,NOTEQUAL
.00000012  4DE0  C022      00000022  32          BAS   R14,SAME
.00000016  47F0  C01E      0000001E  33          B     CONTINUE
.0000001A
.0000001A  4DE0  C022      00000022  35          BAS   R14,DIFFERENT
.0000001E  36  CONTINUE DC    0H
```



Sample Compare and Jump Usage

```
. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.
.
.
.0000000C ECAB 0007 7076          0000001A   31          CRJ    R10,R11,7,NOTEQUAL
.00000012 A7E5 0008          00000022   32          BRAS   R14,SAME
.00000016 A7F4 0004          0000001E   33          BRC    15,CONTINUE
.0000001A          0000001E   34 NOTEQUAL DC    0H
.0000001A A7E5 0004          00000022   35          BRAS   R14,DIFFERENT
.0000001E          0000001E   36 CONTINUE DC    0H
```

```
. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.
.
.
.0000000C ECAB 0007 7076          0000001A   31          CRJNE  R10,R11,NOTEQUAL
.00000012 A7E5 0008          00000022   32          JAS    R14,SAME
.00000016 A7F4 0004          0000001E   33          J      CONTINUE
.0000001A          0000001E   34 NOTEQUAL DC    0H
.0000001A A7E5 0004          00000022   35          JAS    R14,DIFFERENT
.0000001E          0000001E   36 CONTINUE DC    0H
```

“Should Not Occur” Abends

- You can make a branch abend only if taken.
 - Useful while debugging or for “should not occur” logic errors.
 - My favorite technique was to ZAP (or otherwise set) the last bit of the branch displacement ON, resulting in PIC 006 only when branch taken because of invalid displacement.
- There are no invalid jump offsets. However, you can still make a jump abend only if taken.
 - Result is PIC 001 only if branch taken.

```
CLI    0(R1),X'FF'
JE     *+2
```

```
End of table ?
Abend - logic error
```

Loading the Address of an Area Within Your Program

- Without a base register, the LA instruction will not work.
- On machines with z/Architecture (and N3 ESA/390):

```
LARL Rx,ADDRESS
```

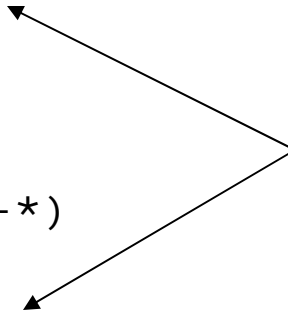
- There is no LAR instruction. Therefore, on older machines, you must use other techniques. Some examples:

```
1. BASR Rx,0  
   AHI Rx,ADDRESS-*  
   LA Rx,0(,Rx)
```

```
2. CNOP 0,4  
   JAS Rx,*+8  
   DC A(ADDRESS-*)  
   AL Rx,0(,Rx)  
   LA Rx,0(,Rx)
```

```
3. L Rx,=A(ADDRESS)
```

LA necessary if high-order bit must be off.
(Can't be used for R0!)



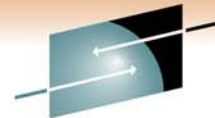
Establishing Temporary Addressability

- You might need temporary addressability, especially when using certain IBM macros (such as those for TSO/E.).
- Be sure to explicitly denote the USING range. The default range of 4K can lead to USING overlap warnings from the assembler.

```
BASR  Rx,0
USING (*,TEMPX),Rx
.
. (code needing addressability)
.
TEMPX DC    0H
```

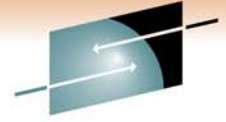
Using **SYS1.MACLIB(IEABRC)** in z/OS

- COPYing IEABRC creates 21 macros and uses OPSYN to intercept and convert branch instructions.
 - Helps with IBM macros that still require program base registers. There are fewer and fewer of these every year.
- There are cases where you will want to selectively enable and disable this conversion. Prior to z/OS 1.10, I had my own macro to do this.
- With z/OS 1.10, the new IEABRCX macro was introduced to “manage” IEABRC. You can dynamically enable/disable the effects of IEABRC as well as save/restore the current settings on a stack (PUSH/POP). Highly recommended!



IEABRCX Usage

. Loc	Object Code	Addr1	Addr2	Stmt	Source	Statement
.0000000C	47F0 C0A0		000000A0	28		B LABEL
.				29		IEABRCX DEFINE
.				366		IEABRCX DISABLE
.				391 *	.	.
.				392 *	.	.
.				393 *	.	.
.00000010	47F0 C0A0		000000A0	394		B LABEL
.				395		IEABRCX ENABLE
.				420		B LABEL
.00000014	A7F4 0046		000000A0	422+		BRC 15,LABEL (B)
.				423		IEABRCX DISABLE
.00000018	47F0 C0A0		000000A0	448		B LABEL
.				449 *	.	.
.				450 *	.	.
.				451 *	.	.
.				452		IEABRCX PUSH
.				453		IEABRCX ENABLE
.				478		B LABEL
.0000001C	A7F4 0042		000000A0	480+		BRC 15,LABEL (B)
.				481		IEABRCX POP
.00000020	47F0 C0A4		000000A4	510		B LABEL

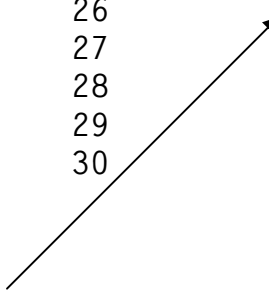


Branch Tables

- There is no indexed jump, so base register is mandatory.
- When handling return codes, the register used for subroutine linkage might be usable as a base.

. Loc	Object Code	Addr1	Addr2	Stmt	Source	Statement
.0000000C	A7E5 003F		0000008A	25	JAS	R14,CALLSERV
.00000010	47FF E004		00000004	26	B	4(R15,R14)
.00000014	A7F4 006D		000000EE	27	J	RETCOD00
.00000018	A7F4 0073		000000FE	28	J	RETCOD04
.0000001C	A7F4 0079		0000010E	29	J	RETCOD08
.00000020	A7F4 007F		0000011E	30	J	RETCOD12

On return from
CALLSERV, R14 points
to the **B** instruction!



Target of EXecute

- The target of an EXecute instruction is fetched into I-cache
- It's best to place it close to the EX itself.

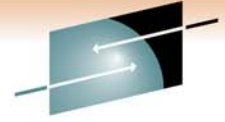
```

. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.0000000C D200 C088 1000 00000088 00000000    25      MVC   WORKFLD(*-*),0(R1)
.00000012 44F0 C00C                0000000C    26      EX    R15,*-6

.00000016 4410 C01E                0000001E    27      EX    R1,DUMMYPAK
.
.
.
.
.0000001A 47F0 C024                00000024    31      B     CONTINUE
.0000001E F270 C0C8 E000 000000C8 00000000    32 DUMMYPAK PACK  DWORD,0(*-*,R14)

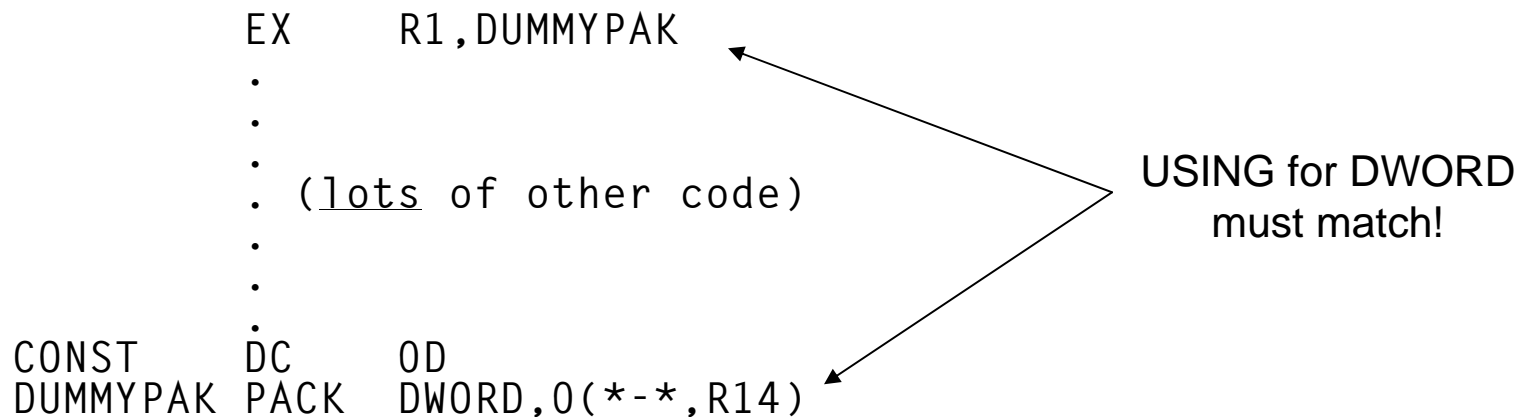
```

- Reference to EX target requires base register coverage.
- There is no EXR instruction. EXRL introduced with z10.
- To maintain similar program layout, establish temporary addressability. Otherwise define the target with constants.



Target of EXecute with Other Constants

- You can place target of EX instruction with constants. But ...
- Be careful! USING(s) in effect when target defined must match the code if an implicit address is used!
- Deferring declaration of a constant might be problematic in general-use macros.



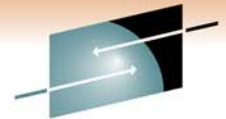
Target of EXecute in Literal Pool

- In a general use macro, you might choose to use a literal.
 - Requires FLAG(NOEXLITW) to avoid ASMA016W warning.
 - Can be controlled via ACONTROL instruction.
- Just like before, USING(s) in effect when target/constant defined must match the code if an implicit address is used!

```
. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.
.                               34 &ExLitMVC SETC 'X''200''(X''D''),'
.                               35          EX   R1,=S(&ExLitMVC,DWORD,0(R14))
.00000024 4410 C18C          0000018C   +          EX   R1,=S(X'200'(X'D'),DWORD,0(R14))
.
.                               . (other code)
.
.0000018C D200C0D0E000          56                               =S(X'200'(X'D'),DWORD,0(R14))
```

- Some convenient “ExLit” values:

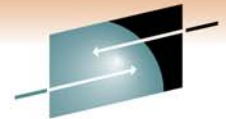
```
&ExLitMVC SETC 'X''200''(X''D''),' MVC instruction w/zero length
&ExLitCLC SETC 'X''500''(X''D''),' CLC instruction w/zero length
&ExLitTR  SETC 'X''C00''(X''D''),' TR  instruction w/zero length
&ExLitTRT SETC 'X''D00''(X''D''),' TRT instruction w/zero length
&ExLitPACK SETC 'X''200''(X''F''),' PACK instruction w/zero length
```



Target of EXecute in Location Counter

- If your program is appropriately structured, use LOCTR to define the EX target with other constants.
- This option ensures current USINGs are honored!
- This is how we handle EX in most of our products.

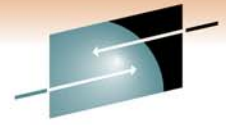
```
. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.00000016 4410 C1A0              000001A0    30          EX    R1,DUMMYPAK
.000001A0              00000197 000001A6    31 DATA  LOCTR ,
.000001A0 F270 C0C8 E000 000000C8 00000000    32 DUMMYPAK PACK  DWORD,0(*-*,R14)
.0000001A              00000000 000001A6    33 CODE   LOCTR ,
.              *              .
.              *              . (more instructions)
.              *              .
```



Execute Relative Long

- EXRL intended as a direct substitute for EX.
- Supports long-standing recommendations, to put the target of the EX near the EX instruction itself, for programs without code base register coverage.
 - Hopefully, same I-Cache line or one to be fetched soon.
- Available on System z10 and higher processors.
- If you can use this, I'm jealous. Too new for our software!

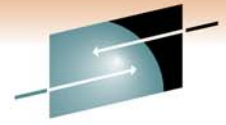
```
.00000000 C610 0000 0005          0000000A    25          EXRL  R1,DUMMYPAK
.                                     26 *          .
.                                     27 *          . (a few other instructions)
.                                     28 *          .
.00000006 A7F4 0005          00000010    29          J      AROUND
.0000000A F270 C0B8 E000 000000B8 00000000    30 DUMMYPAK PACK  DWORD,0(*-*,R14)
.00000010          31 AROUND  DC    OH
```



Executing a Jump

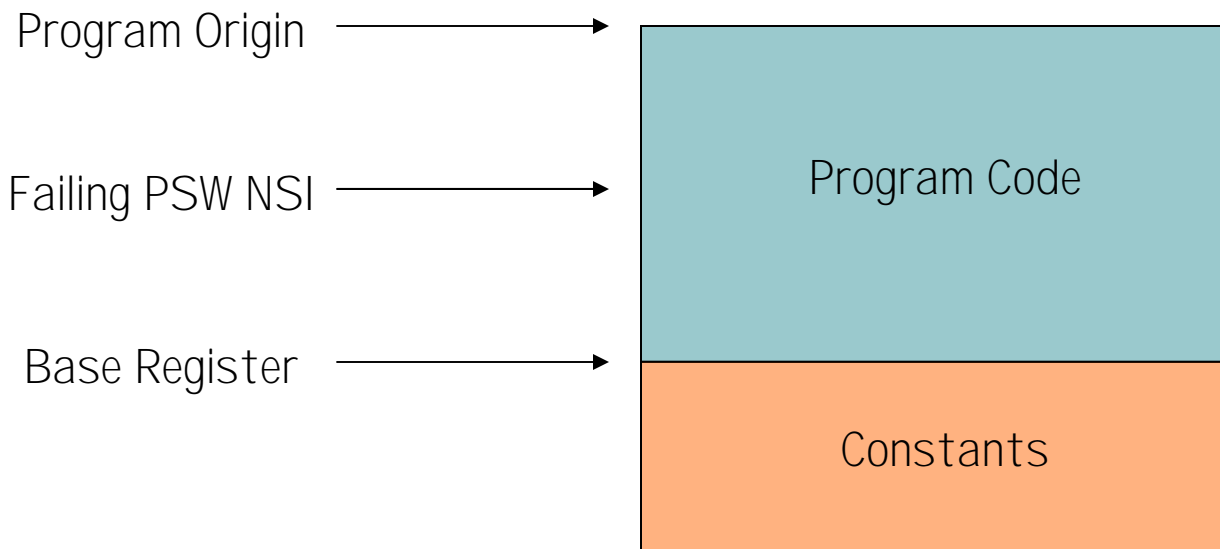
- When a jump is the target of an EX or EXRL instruction, the jump target address is calculated by adding the immediate value to the address of the jump itself. The current instruction address in the PSW is not considered.
- This is both necessary and convenient, because that's how the jump instruction will be assembled in storage.
- Same applies to new **Compare and Jump** instructions.

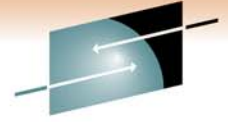
```
. Loc      Object Code      Addr1      Addr2      Stmt  Source Statement
.00000016 4310 C12D              0000012D    30      IC      R1,JUMPMASK
.0000001A 4410 C128              00000128    31      EX      R1,JUMP2LBL
.
. (more code)
.00000048              33 LABEL    DC      0H
.
. (more code & constants)
.00000128 A704 FF90          00000048    51 JUMP2LBL JNOP LABEL
```



Base Register Origin

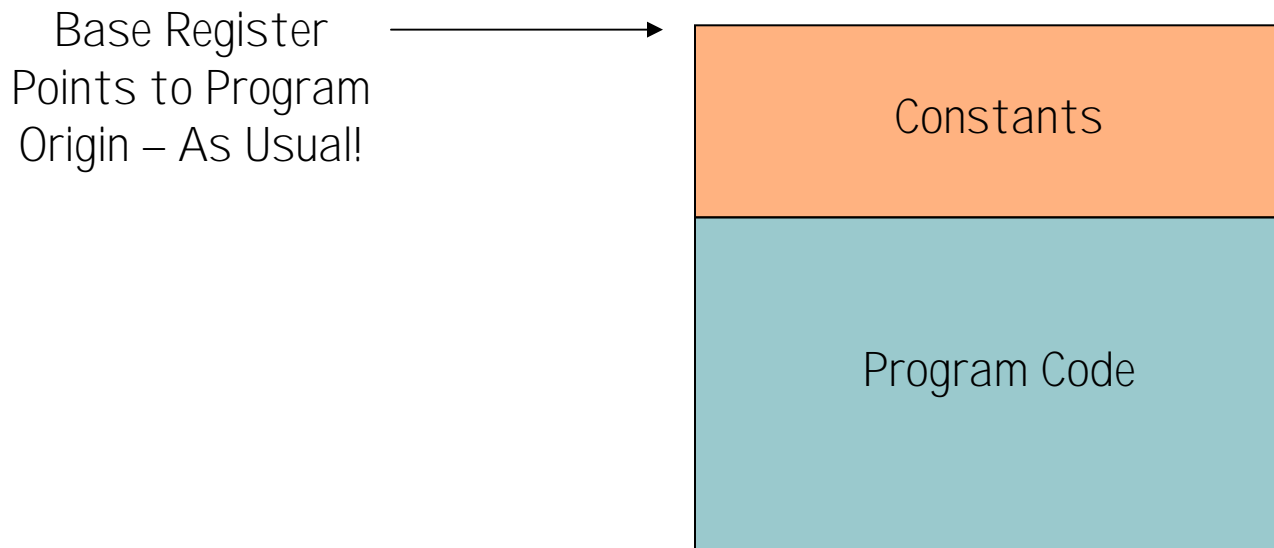
- Base register pointing to beginning of program is convenient, especially if the program is aligned on a cache line or page boundary.
- Base register pointing to middle of program is far less so – making post-mortem analysis more difficult.
 - I disliked having to use negative offsets all the time.

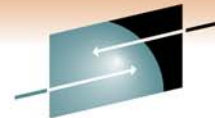




Putting Constants First

- Base register points to beginning of program.
- Constants (and working storage for non-reentrant programs) physically exist at the beginning and are “covered” by the base.
- Best of both worlds solution!





Putting Constants First

. Loc	Object Code	Addr1	Addr2	Stmt	Source	Statement
.00000000		00000000	00000036	1	MYPGM	CSECT ,
.00000000	C0F4 0000 0013		00000026	2		JLU CODE
.00000006	C5A885C381A38388			3		DC CL10'EyeCatcher'
.00000010		00000010	00000036	4	DATA	LOCTR ,
.00000026		00000026	00000036	5	CODE	LOCTR ,
.00000026	18CF			6		LR R12,R15
.		R:C 00000000		7		USING MYPGM,R12
.				8	*	.
.				9	*	. (code here)
.				10	*	.
.00000028	5910 C018		00000018	11		C R1,=F'1E4'
.0000002C	4120 C01C		0000001C	12		LA R2,=C'An example'
.00000030	E330 C010 0004		00000010	13		LG R3,=AD(123)
.				14	*	.
.				15	*	.
.00000008		00000008	00000036	16	DATA	LOCTR ,
.00000010				17		LTORG ,
.00000010	00000000000000007B			18		=AD(123)
.00000018	00002710			19		=F'1E4'
.0000001C	C1954085A7819497			20		=C'An example'
.				21	*	.
.				22	*	. (other constants)
.				23	*	.
.				43		END ,

Relative-Immediate Support in z/OS V1R7 Binder

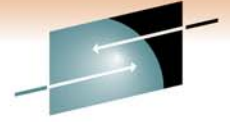


- Support all relative-immediate instructions (BRAS [JAS], BRASL [JLAS], LARL, etc.) with one external symbol in their operands.
- New RI-con RLD item (GOFF only).
- Support similar arithmetic calculations as A-cons.
- Support cross-class references (instruction and target in same load segment but different classes).
- New z/OS V1R7 Program Object format.
 - Works only on z/OS V1R7 and later binders.
 - Still called PO4. Confusing. ☹
- **NOTE:** z/OS 1.7 is out of service.

Relative-Immediate Support in z/OS V1R8 Binder



- Support cross-compile unit references in relative-immediate in traditional load modules and object modules.
- Support new encoding of RLD data flag (xBTTLLxx):
 - 11100 = two-byte relative-immediate.
 - 11110 = four-byte relative-immediate.
- Support cross-segment references in relative-immediate instructions in program objects.
 - Two-byte are allowed in a single segment.
 - Four-byte are allowed across segment except if either segment is RMODE(64).
- New Program Object format PO5.
- **NOTE:** z/OS 1.8 is out of service.



SHARE
Technology • Connections • Results

The End

SHARE
in Anaheim
2011