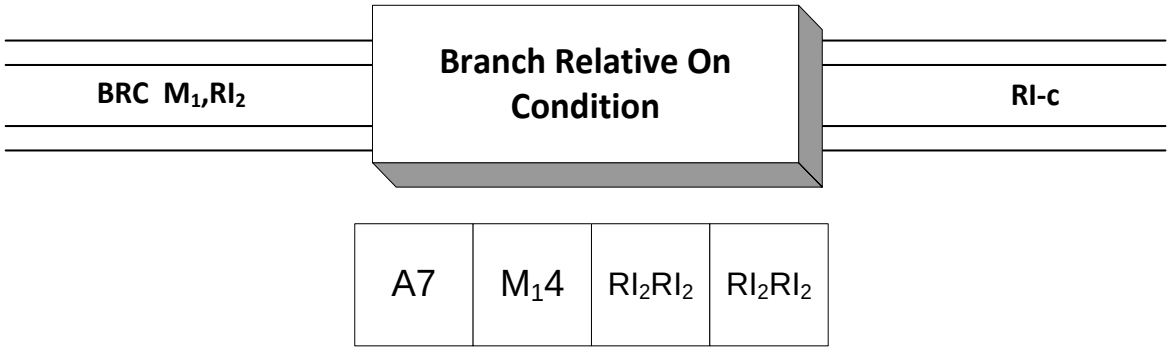


BRC is an RI-c instruction whose use is similar to **BR** (Branch on Condition) but provides a different mechanism for branching (relative branching) that doesn't require the use of a base register. As a result, **BRC** was designed to provide some register relief by replacing **BC** in many instances. The 16-bit 2's complement integer, RI_2, describes the number of halfwords from the beginning byte of the **BRC** instruction to the target address. In other words, the branch is relative to the address of the instruction, and not measured from the base address of some register. As a result, the target address doesn't have to lie in the range of a USING statement – this can provide some register relief. RI_2 satisfies the constraint $-2^{15} \le RI_2 \le 2^{15}-1$, or $-32768 \le RI_2 \le 32767$. Since this integer represents a number of halfwords, the relative offset to the target address can be as far as −65536 and +65534 bytes away, allowing us to branch forward or backward. Why does RI_2 represent a number of halfwords instead of bytes? Since all machine instructions have an even number of bytes, we never need to branch to an odd-numbered address. Letting RI_2 represent a number of halfwords effectively doubles the range of the target address.

BRC has a 12-bit opcode – A74.

As in the case of BC, M_1 is a four-bit mask that describes the conditions on which the branch will occur.

There are four possible values for the condition code:

Condition Code	Meaning
00	Zero or Equal
01	Low or Minus
10	High or Plus
11	Overflow

When constructing a mask for Operand 1, each bit (moving from the high-order bit to the low-order bit) represents one of the four conditions in the following order: Zero/Equal, Low/Minus, High/Plus, Overflow. Consider the following instruction,

```
BRC   8,THERE
```

The first operand,"8", is a decimal self-defining term and represents the binary mask B'1000'. Since the first bit is a 1, the mask indicates that a branch should occur on a zero or equal condition. Since the other bits are all 0, no branch will be taken on the other conditions. The first operand could be designated as any equivalent self-defining term. For example, the following instruction is equivalent to the one above.

```
BRC   B'1000',THERE
```

There are two sets of extended mnemonics for relative branches designed to replace the awkward construction of having to code a mask. The extended mnemonics are easier to code and read. The first set uses "BR" followed by a suffix that indicates the type of branch.

```
BRE   BRANCH RELATIVE Equal          BRNE   BRANCH RELATIVE Not Equal
BRZ   BRANCH RELATIVE Zero           BRNZ   BRANCH RELATIVE Not Zero
BRL   BRANCH RELATIVE Low            BRNL   BRANCH RELATIVE Not Low
BRM   BRANCH RELATIVE Minus          BRNM   BRANCH RELATIVE Not Minus
BRH   BRANCH RELATIVE High           BRNH   BRANCH RELATIVE Not High
BRP   BRANCH RELATIVE Positive       BRNP   BRANCH RELATIVE Not Positive
BRU   BRANCH RELATIVE Unconditional
```

The second set of mnemonics replaces the "BR" with "J" for jump. This set also includes a no-operation mnemonic.

```
JE      Jump Equal          JNE      Jump Not Equal
JZ      Jump Zero           JNZ      Jump Not Zero
JL      Jump Low            JNL      Jump Not Low
JM      Jump Minus          JNM      Jump Not Minus
JH      Jump High           JNH      Jump Not High
JP      Jump Positive       JNP      Jump Not Positive
JO      Jump Overflow       JNO      Jump No Overflow
JNOP    No Operation        J        Unconditional Jump
```

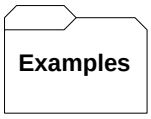
In most cases, there are three ways to code a relative branch. For example, the following three instructions are equivalent and generate the same object code:

```
BRC  4,THERE      Mask = Low
BRL  THERE
JL   THERE
```

The table below indicates the possible mask values and the equivalent Jump extended mnemonics.

Eq/Low	Low/Min	High/Plus	Overflow	Decimal Condition	Extended Mnemonic
0	0	0	0	0	JNOP
0	0	0	1	1	JO
0	0	1	0	2	JH, JP
0	0	1	1	3	No mnemonic
0	1	0	0	4	JL, JM
0	1	0	1	5	No mnemonic
0	1	1	0	6	No mnemonic
0	1	1	1	7	JNE, JNZ

Eq/Low	Low/Min	High/Plus	Overflow	Decimal Condition	Extended Mnemonic
1	0	0	0	8	JE, JZ
1	0	0	1	9	No mnemonic
1	0	1	0	10	No mnemonic
1	0	1	1	11	JNL, JNM
1	1	0	0	12	No mnemonic
1	1	0	1	13	JNH, JNP
1	1	1	0	14	No mnemonic
1	1	1	1	15	J



Some unrelated BRC's organized into equivalent groupings of three:

```
BRC   8,THERE      BRANCH RELATIVE ON CONDITION EQUAL
BRE   THERE        BRANCH RELATIVE ON CONDITION EQUAL
JE    THERE        JUMP EQUAL

BRC   7,THERE      BRANCH RELATIVE ON CONDITION NOT EQUAL
BRNE  THERE        BRANCH RELATIVE ON CONDITION NOT EQUAL
JNE   THERE        JUMP NOT EQUAL

BRC   15,THERE     BRANCH RELATIVE UNCONDITIONALLY
BRU   THERE        BRANCH RELATIVE UNCONDITIONALLY
J     THERE        JUMP UNCONDITIONALLY
```

Tips

1. Use Jumps to simplify relative branches. Replacing BCs and their related mnemonics with Jumps may provide some register relief and allow more of your data to be covered by a base register.
2. Try coding the IEABRCX system macro near the top of a program that contains branch mnemonics (BCs). You can do this by simply inserting "IEABRCX DEFINE" as a macro in the program. Assemble the program and look at the object code for the branch instructions in the program before and after coding the macro. IEABRCX will cause the assembler to generate relative branch instructions for most of the standard branch instructions.
3. Read about IEABRCX in z/OS MVS Programming - Assembler Services Reference: IAR-XCT http://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.ieaa900/toc.htm
4. There's not much of a drawback to replacing BCs with BCRs. The instruction sizes are identical so the program won't change size, and you will get the benefit of not having to use a base register to cover your target addresses. The one disadvantage is that relative branch instructions don't allow indexing.