# Jumping Into Branches

This paper will investigate the many options we have for engineering the flow of control in our programs. We discuss the native and mnemonic branch and jump instructions that can be employed, and consider the mechanisms used by the machine to implement both types of instructions. We consider branch instructions that are used to create loops and those used to transfer control between modules. We also investigate some non-branching instructions that use relative addressing as a way of avoiding those pesky base/displacement addresses.

## In the Beginning…

The original (pre-ESA-390) workhorse branching instructions were Branch on Condition (BC) and Branch on Condition Register (BCR). Of these two, we first consider the more commonly used Branch on Condition (BC). This RX instruction (explicit format: M1,D2(X2,B2) ) uses the four-bit area (normally devoted to R1 in most RX instructions) as a binary mask, M1. Each 1-bit in the mask represents a condition under which we are willing to branch. Correspondingly, each 0-bit in the mask represents a condition under which we do not want to branch. The meaning of each bit is listed below.

Numbering bits from left to right:

```
Bit 0:    equal or zero
Bit 1:    low or minus
Bit 2:    high or plus
Bit 3:    overflow or ones
```

With four bits, there are 16 combinations of conditions for the mask. The PSW's condition code (two bits) represents one of the four possible conditions: 0 – equal/zero, 1 – low/minus, 2 – high/plus, and 3 – overflow/ones. If the mask contains a 1-bit in the condition represented by the condition code, the branch will occur, otherwise flow of controls picks up with the next sequential instruction. For example, if you coded the following:

```
          BC   8,THERE
```

Operand 1 becomes the binary mask, in this case, 8 = B'1000', which indicates that we will branch only if the condition code is equal (zero). If the PSW's condition code happened to be 0 (equal) when we executed the branch above, the branch would occur. On the other hand, if the condition code were 1 (low), 2 (high), or 3 (overflow) when we executed the branch, execution would fall through and continue with the next instruction. The mask insures that we will branch only on an equal (zero) condition.

As another example, suppose you coded the following:

```
          BC   11,THERE
```

In this case, 11 = B'1011', indicates that we should branch on equal (zero), high (plus), or overflow (ones). Another way to describe this is to say we will branch on anything but low, or more simply, branch not low. If the PSW condition code is 0 (equal), 2 (high), or 3 (overflow), the branch will occur.

These ideas explain why the following is an unconditional branch:

```
BC   15,THERE
```

In this case, 15 = B'1111', indicates we should branch under all possible conditions, so the branch will occur no matter the contents of the PSW.

# Extended Mnemonics for BC

It is inconvenient to have to build and interpret binary masks, so instead of coding native BC instructions, we can use "extended mnemonics" instead. The assembler will automatically convert our extended mnemonics to equivalent BC instructions. The following table lists all the extended mnemonics and their corresponding native equivalent BC instructions. For example, the table shows that the two branch instructions below are equivalent:

```
BC    13,THERE
BNH   THERE
```

Extended mnemonics exist for all reasonable values of four bit patterns. There are a few values for which there is no extended mnemonic, for example,

```
BC   9,THERE
```

With a mask of 9 = B'1001', we would branch on Equal (zero), or Overflow (ones). This is a rather unlikely combination, so there doesn't exist an equivalent mnemonic for this mask.

| **Machine Instruction** | **Extended Mnemonic** | **Meaning** |
|---|---|---|
| BC 15,D2(X2,B2) | B | Branch Unconditionally |
| BC 0,D2(X2,B2) | NOP | No Operation |
| Used after Compare Instructions(A:B): | | |
| BC 2,D2(X2,B2) | BH | Branch on A High |
| BC 4,D2(X2,B2) | BL | Branch on A Low |
| BC 8,D2(X2,B2) | BE | Branch on A Equal B |
| BC 13,D2(X2,B2) | BNH | Branch on A Not High |
| BC 11,D2(X2,B2) | BNL | Branch on A Not Low |
| BC 7,D2(X2,B2) | BNE | Branch on A Not Equal B |
| Used after Arithmetic instructions: | | |
| BC 1,D2(X2,B2) | BO | Branch On Overflow |
| BC 2,D2(X2,B2) | BP | Branch On Plus |
| BC 4,D2(X2,B2) | BM | Branch On Minus |
| BC 8,D2(X2,B2) | BZ | Branch On Zero |
| BC 14,D2(X2,B2) | BNO | Branch On Not Overflow |
| BC 13,D2(X2,B2) | BNP | Branch On Not Plus |
| BC 11,D2(X2,B2) | BNM | Branch On Not Minus |
| BC 7,D2(X2,B2) | BNZ | Branch On Not Zero |
| Used after Test Under Mask: | | |

```
BC     1,D2(X2,B2)          BO                Branch if Ones
BC     4,D2(X2,B2)          BM                Branch if Mixed
BC     8,D2(X2,B2)          BZ                Branch if Zeros
BC     14,D2(X2,B2)         BNO               Branch if Not Ones
BC     11,D2(X2,B2)         BNM               Branch if Not Mixed
BC     7,D2(X2,B2)          BNZ               Branch if Not Zeros
```

## Branching Mechanism for BC

For all BC instructions, operand 2 is a target address which is converted to a base/displacement address. Consider for example,

```
BC   15,THERE
```

THERE is a symbolic address in the program that is converted to a base/displacement in object code.

For the related BC extended mnemonics, the single operand is also a symbolic address which is converted to the equivalent base/displacement address in memory. For example in,

```
BNE   THERE
```

the symbol THERE is converted to an equivalent base/displacement format in memory. The main point to be noted is that both forms of these branch instructions require the use of a base register to facilitate the branch.

## Branch On Condition Register

The related RR instruction for branching is Branch on Condition Register. This instruction is used less frequently than BC simply because it requires an extra register. In this version of a branch, operand 1 is again treated as a mask, but operand 2 is a register containing the target address. As in the case of BC, the PSW's condition code is checked against the mask in the branch instruction to see if the corresponding mask bit has been set to one. If so, the branch occurs. Here's an example:

```
LA    5,THERE
BCR   8,5
```

Assuming the condition code is 0 (equal), the branch would occur since 8 = B'1000', and the 1 indicates we should branch on equal (zero). Since register 5 contains the address of THERE, execution would continue at that address. While BCR doesn't use a base/displacement address directly, it does require the use of a register to facilitate the branch.

The BCR instruction comes with its own set of extended mnemonics which are listed below. As with BC, some masked instructions do not have equivalent mnemonics.

| Machine Instruction | Extended Mnemonic | Meaning |
|---|---|---|
| BCR 15,R2 | BR | Branch Unconditionally |
| BCR 0,R2 | NOPR | No Operation |

Used after Compare Instructions(A:B):

| | | |
|---|---|---|
| BCR 2,R2 | BHR | Branch on A High |
| BCR 4,R2 | BLR | Branch on A Low |
| BCR 8,R2 | BER | Branch on A Equal B |
| BCR 13,R2 | BNHR | Branch on A Not High |
| BCR 11,R2 | BNLR | Branch on A Not Low |
| BCR 7,R2 | BNER | Branch on A Not Equal B |

Used after Arithmetic instructions:

| | | |
|---|---|---|
| BCR 1,R2 | BOR | Branch On Overflow |
| BCR 2,R2 | BPR | Branch On Plus |
| BCR 4,R2 | BMR | Branch On Minus |
| BCR 8,R2 | BZR | Branch On Zero |
| BCR 14,R2 | BNOR | Branch On Not Overflow |
| BCR 13,R2 | BNPR | Branch On Not Plus |
| BCR 11,R2 | BNMR | Branch On Not Minus |
| BCR 7,R2 | BNZR | Branch On Not Zero |

Used after Test Under Mask:

| | | |
|---|---|---|
| BC 1,R2 | BOR | Branch if Ones |
| BC 4,R2 | BMR | Branch if Mixed |
| BC 8,R2 | BZR | Branch if Zeros |
| BC 14,R2 | BNOR | Branch if Not Ones |
| BC 11,R2 | BNMR | Branch if Not Mixed |
| BC 7,R2 | BNZR | Branch if Not Zeros |

As you might have noticed, the equivalent RR mnemonic can be obtained from the RX mnemonic by adding “R” at the end, so BNER is the RR equivalent of the RX BNE mnemonic.

## An Advantage and Disadvantage to Using BC to Branch

Since BC is an RX instruction, it offers us the opportunity to use indexing as part of the branch address. In practice this is rarely done, so this advantage is a relatively minor one. There is one interesting use of this feature when calling and returning from a module that sets a return code in R15. Let’s assume we will call SUBPROG from a main program and SUBPROG sets the return code to 0, 4, 8 or 12. We further assume that when our main program resumes control, it needs to branch to RTN1 when RC=0, RTN2 when RC=4, RTN3 when RC=8, and RTN4 when RC=12. The main program builds a branch table consisting of consecutive branch instructions to those routines. We use R15 as an index register to jump into the “middle” of the branch table at the correct position. Here’s the code:

```
                LINK  EP=SUBPROG        CALL THE SUBPROG
                B     BRANCHTAB(R15)    BRANCH INTO THE TABLE
BRANCHTAB       EQU   *
                B     RTN1
                B     RTN2
                B     RTN3
                B     RTN4
```

Suppose SUBPROG executes and sets R15 to 8 before returning. Upon return, the main program executes B BRANCHTAB(R15) . Since R15 contains 8, the effective indexed address is BRANCHTAB+8. Each branch instruction in the table occupies 4 bytes, so BRANCHTAB+8 references the third branch instruction in the table, which immediately sends us to RTN3. In a similar way, R15 = 0 will send us to RTN1, R15=4 will send us to RTN2, and R15=12 sends us to RTN4. Using the return code register as an index allows us to avoid a series of compare operations upon returning from SUBPROG.

The main disadvantage to using a BC operation is that it requires a base register for representing the target address in base/displacement format. For long programs, we may need to devote several registers to cover the symbolic target addresses that are scattered through our executable code. This disadvantage is significant given the scarcity of available registers. A second disadvantage to BC is that the target address must be within a 4095 byte displacement of the base register address. For large programs, this too, can be a significant problem which requires us to devote more registers to creating base/displacement addresses.

## A Different Mechanism for Branching

To address some of the limitations of BC, in 1990 IBM's ESA/390 architecture introduced Branch Relative on Condition (BRC) and Branch Relative on Condition Long (BRCL) instructions that contain longer displacements and use a technique of branching that doesn't require a base register at all. In the case of BRC, the instruction is of a new instruction type, RI, and contains a 16-bit 2's complement integer as part of the instruction. To facilitate the branch, the machine doubles this 2's complement integer, and adds it to the beginning address of the branch instruction to determine the target address, bypassing the need for a base register. With this technique, we can move forward or backward from the branch instruction. With 16 bits, 2's complement integer values range from −32768 to +32767. Doubling (all instructions have lengths which are multiples of 2) allows the relative offset to the branch target to lie as many as −65536 and +65534 bytes away – all this without a base register.

The BRCL instruction is of a new instruction type, RIL, and contains a 2's complement integer that provides a relative offset that ranges from −2,147,483,648 to 2,147,483,647. When doubled we can branch forward or backward four billion bytes. That should be sufficient for most programs!

Coding a BRC or BRCL is similar to coding a BC. Operand 1 represents a 4-bit mask that describes the conditions under which we will branch. Operand 2 is a target address. The assembler computes the relative offset that is needed automatically.

Here's an example use of a BRC instruction:

```
CLC       A,B
BRC       8,RTN4
```

First we compare two fields to set the condition code, and then we test the condition code with BRC. The mask is 8=B'1000', and we take the branch if the condition code is equal (zero). The BRC could be replaced by BRCL and the program should run identically, although BRCL is a six-byte instruction while BRC is only a four-byte instruction since it contains a smaller integer.

As in the case of BC, IBM provides extended mnemonics for these relative branch instructions. The letter "R" follows the "B" in the extended mnemonics to indicate a relative branch will be used. To remind us that the mechanism for relative branching is different, there is a second set of extended relative branch mnemonics called "jumps" and the mnemonics begin with "J". In most cases, an "R" can be inserted into the BC extended mnemonics to obtain the BRC mnemonic. One exception is BRU. The "U" is added because BR was already an existing instruction. In any case, there are two sets of extended mnemonics for relative branches – mnemonics that begin with "BR" and a second set that uses "J". As a good coding practice, I recommend using the extended mnemonics that begin with "J".

For example,

```
BC      8,THERE    NATIVE BRANCH ON CONDITION
BE      THERE      EQUIVALENT MNEMONIC FOR CONDITION 8

BRC     8,THERE    RELATIVE BRANCH INSTRUCTION
BRE     THERE      B-VERSION MNEMONIC FOR CONDITION 8
JE      THERE      J-VERSION MNEMONIC FOR BRC
```

The table below lists the BRC instructions and their equivalent mnemonics.

| RI Instruction | Mnemonic | Mask | Meaning |
|---|---|---|---|
| BRC | JC | M1 | Conditional Branch |
| BRU | J | 15 | Unconditional Branch |
| BRNO | JNO | 14 | Branch if Not Ones |
| | | | Branch if No Overflow |
| BRNH | JNH | 13 | Branch if Not High |
| BRNP | JNP | 13 | Branch if Not Plus |
| BRNL | JNL | 11 | Branch if Not Low |
| BRNM | JNM | 11 | Branch if Not Minus |
| | | | Branch if Not Mixed |
| BRE | JE | 8 | Branch if Equal |
| BRZ | JZ | 8 | Branch if Zero |
| BRNZ | JNZ | 7 | Branch if Not Zero |
| BRNE | JNE | 7 | Branch if Not Equal |
| BRL | JL | 4 | Branch if Low |
| BRM | JM | 4 | Branch if Minus |
| | | | Branch if Mixed |

| | | | |
|---|---|---|---|
| BRH | JH | 2 | Branch if High |
| BRP | JP | 2 | Branch if Plus |
| BRO | JO | 1 | Branch if Ones |
| | | | Branch if Overflow |
| | JNOP | 0 | No Operation |

For branches distances greater than +/- 64K, the table below lists the BRCL instructions and their equivalent mnemonics.

| RI Instruction | Mnemonic | Mask | Meaning |
|---|---|---|---|
| BRCL | JLC | M1 | Conditional Branch Long |
| BRUL | JL | 15 | Unconditional Branch Long |
| BRNOL | JLNO | 14 | Branch if Not Ones Long |
| | | | Branch if No Overflow Long |
| BRNHL | JLNH | 13 | Branch if Not High Long |
| BRNPL | JLNP | 13 | Branch if Not Plus Long |
| BRNLL | JLNL | 11 | Branch if Not Low Long |
| BRNML | JLNM | 11 | Branch if Not Minus Long |
| | | | Branch if Not Mixed Long |
| BREL | JLE | 8 | Branch if Equal Long |
| BRZL | JLZ | 8 | Branch if Zero Long |
| BRNZL | JLNZ | 7 | Branch if Not Zero Long |
| BRNEL | JLNE | 7 | Branch if Not Equal Long |
| BRLL | JLL | 4 | Branch if Low Long |
| BRML | JLM | 4 | Branch if Minus Long |
| | | | Branch if Mixed Long |
| BRHL | JLH | 2 | Branch if High Long |
| BRPL | JLP | 2 | Branch if Plus Long |
| BROL | JLO | 1 | Branch if Ones Long |
| | | | Branch if Overflow Long |
| | JLNOP | 0 | No Operation Long |

## What's a Poor Programmer to Do?

So far we've covered the basic instructions for moving from here to there. There are different options for creating loops, and for branching between modules. We will cover those instructions shortly, but what do we make of the choices so far? Does it matter if we choose relative jumps over their base/displacement counterparts? Here, we are drifting a bit into matters of style and design.

Older programs are more likely to use base/displacement branches, and if a program is working, there's little to gain by moving to relative branching in new code you might add. Occasionally though, a program will suffer from code-bloat, and it becomes difficult to add more code or variables without encountering addressability errors. In that case, one possibility for relief is to change to relative branching and modify the USING statements to include only the data in your program, not the code itself. This offers the possibility of reducing the number of required base registers and providing more

room for the data. This process of converting to relative branching presents a few subtle challenges, and I will cover those in a separate paper.

For new code, there's no reason not to use relative branching – it's a better choice, if for no other reason than it offers the possibility of needing fewer base registers, so I say "jump in".

## Getting Loopy

Pre-ESA/390, the instructions used for manipulating "counted" loops included Branch on Count (BCT), Branch on Index High (BXH), and Branch on Index Less Than or Equal (BXLE). Starting with ESA/390, these instructions have related companions: Branch Relative on Count (BRCT), Branch Relative on Index High (BRXH), and Branch Relative on Index Less Than or Equal (BRXLE). These instructions avoid the use of base/displacement addresses by using the same mechanism we described for the jump instructions above. As in the case of BC, there are extended mnemonics for these branches as well:

| Machine Instruction | Extended Mnemonic | Meaning |
|---|---|---|
| BRCT R1,I2 | JCT | Jump on Count |
| BRXH R1,I2 | JXH | Jump on Index High |
| BRXLE R1,I2 | JXLE | Jump on Index Less Than or Equal |

The instructions above work with 32-bit registers, but there are "Grande" companions that perform similar operations on the 64-bit registers:

| Machine Instruction | Extended Mnemonic | Meaning |
|---|---|---|
| BRCTG R1,I2 | JCTG | Jump on Count Grande |
| BRXHG R1,I2 | JXHG | Jump on Index High Grande |
| BRXLEG R1,I2 | JXLEG | Jump on Index Less Than or Equal Grande |

## Calling Other Modules

Branch and Save (BAS), the old workhorse instruction used to move between modules, has two relative branch companion instructions with appropriate mnemonics:

| Machine Instruction | Extended Mnemonic | Meaning |
|---|---|---|
| BRAS R1,I2 | JAS | Jump and Save |
| BRASL R1,I2 | JASL | Jump and Save Long |

# Referring to Addresses in Your Code

Besides using a BC or BCR to refer to an area of executable code (and generating a base/displacement address), occasionally you need to load the address of something inside your executable code. If you are using Load Address (LA) for this purpose, you are forced to devote part of a USING range to cover your executable code, and this reduces the amount of the USING range left for the data in your program. To avoid this problem, you can use Load Address Relative Long (LARL), which uses a relative address constant (instead of a base/displacement) to represent the distance (in halfwords) from the LARL instruction to the target. The constant is doubled, added to the address of the LARL, and stored in operand 1. The effect is the same as using a LA instruction, but we no longer have to devote a base register to covering the target address. The target address can be +/- 64K bytes away from the LARL. Because the constant is doubled, only even-numbered addresses can be generated. If you are trying to load an odd address, you will have to adjust the register by 1.

LARL offers us new ways to load base registers. Unlike LA which is limited to displacements of 0 to 4**0**95, LARL can work with much larger values when loading base registers. Here's one method for establishing and loading base registers 12, 11 and 10:

```
000004 0DC0                            36          BASR  R12,R0
               R:CBA  00006            37          USING BASEADR,R12,R11,R10
                      00006            38 BASEADR  EQU   *
000006 C0B0 0000 0800       01006      39          LARL  R11,BASEADR+4096
00000C C0A0 0000 0FFD       02006      40          LARL  R10,BASEADR+8192
```

We load base register 12 with BASR and declare R12, R11, and R10 as the base registers for this program. LARL is not constrained by a 4k displacement and easily builds the address that is 4k larger than the base address in R12. It repeats this process for R10.

Ask yourself "Why does the following code not work?" – It assembles without error but executes with an S0C4. Look at the assembled code if you are stumped:

```
000004 0DC0                            36          BASR  R12,R0
               R:CBA  00006            37          USING BASEADR,R12,R11,R10
                      00006            38 BASEADR  EQU   *
000006 41B0 B000            01006      39          LA    R11,BASEADR+4096
00000A 41A0 A000            02006      40          LA    R10,BASEADR+8192
```

Answer: R12 is correctly loaded as before, and R12, R11, and R10 are declared to be base registers for the program. The BASR/USING combination effectively establishes addressability with R12. That means we can use symbols in the 4K range of R12: *+0, *+1,…,*+4095 where * refers to the base address for R12 which is also BASEADR. Unfortunately, BASEADR+4096 is one byte outside the range of R12, so the assembler generates B000 for the base/displacement address R12+4096 when R12 is the only register we have loaded at this point. In other words, in the first LA instruction, we are trying to establish R11 with the correct address (so it can be a base register) while trying to use it as a base register (B000) at the same time. Before B000 can make sense as a base/displacement address, R11 must already contain

the base address.  Similar remarks would apply in the second LA instruction.  But LARL is unconstrained by 4k boundaries, so the first technique listed above works correctly while the second does not.