Op Code	$M_1$4	I_2I_2	I_2I_2	I_2I_2	I_2I_2

The Branch Relative on Condition Long instruction works exactly like BRC but provides a larger two's complement integer to describe the length of the branch, so it has the ability to jump much further forward or backward than BRC. The instruction examines the 2-bit condition code in the PSW and branches (or not) based on the value it finds. Operand 1 is a self-defining term which represents a 4-bit mask M_1 (binary pattern) indicating the conditions under which the branch should occur. Operand 2 is the target address to which the branch will be made if the condition indicated in Operand 1 occurs. Rather than representing the target as a base/displacement address, the number of half bytes from the current instruction to the target address is computed as a two's complement integer and stored in the instruction as I_2.

BRCL is similar to BC except for the mechanism of specifying the target address –this is the address which replaces the current PSW instruction address field. In the case of BRCL, the target address is computed by doubling the two's complement integer represented in I_2, and adding the result to the address of the current instruction (not the PSW instruction address). If the condition code has one of the values specified in the mask, the instruction address in the PSW is replaced with this "relative" address. Since I_2 is a 32-bit two's complement integer that represents a number of half bytes, this instruction can be used to branch forward $(2^{32} – 1) \times 2$ (approximately 4G) and backward $2^{32} \times 2$ (approximately -4G).

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,

```
BRCL  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.

```
BRCL  B'1000',THERE
```

When relative branching was introduced, new sets of extended mnemonics were also developed to replace the awkward construction of having to code a mask. The extended mnemonics are converted to BRC's and are easier to code and read than using masks. Here is a list of the branch relative extended mnemonics and the equivalent jump extended mnemonics.

Low/Min	High/Plus	Overflow	Decimal Condition	Extended Mnemonic
0	0	0	0	JLNOP
0	0	1	1	BROL,JLO
0	1	0	2	BRHL,JLH BRPL,JLP
0	1	1	3	NO MNEMONIC
1	0	0	4	BRML,JLL BRLL,JLM
1	0	1	5	NO MNEMONIC
1	1	0	6	NO MNEMONIC
1	1	1	7	BRNEL,JLNE BNZL,JLNZ
0	0	0	8	BREL,JLE BRZL,JLZ
0	0	1	9	NO MNEMONIC
0	1	0	10	NO MNEMONIC
0	1	1	11	BRNLL,JLNL BRNML,JLNM
1	0	0	12	NO MNEMONIC

1	0	1	13	BRNHL,JLNH BRNPL,JLNP
1	1	0	14	NO MNEMONIC
1	1	1	15	BRUL,JLU

Notice that branch relative mnemonics have equivalent jump mnemonics. Simply replacing "BR" with "J" produces the equivalent mnemonic in most cases. NOPs and unconditional branches are the exceptions.

Using extended mnemonics we could replace the previous Branch Relative On Condition instruction (`BRC B'1000',THERE`) with any of the following instructions,

```
BRZ  THERE
JZ   THERE
BRE  THERE
JE   THERE
```

When the assembler processes **BRZ, JZ**, **BRE**, or **JE** it generates the mask as B'1000'. The table below indicates the possible mask values and the equivalent extended mnemonics.

Here is an alternate listing of the extended mnemonics for BRCL followed by the equivalent Jump mnemonics.

Branch Relative on Condition Long

```
BROL   label          Br Rel Long on Overflow       RIL       BRCL  1,label
BRHL   label          Br Rel Long on High           RIL       BRCL  2,label
BRPL   label          Br Rel Long on Plus           RIL       BRCL  2,label
BRLL   label          Br Rel Long on Low            RIL       BRCL  4,label
BRML   label          Br Rel Long on Minus          RIL       BRCL  4,label
BRNEL  label          Br Rel Long on Not Equal      RIL       BRCL  7,label
BRNZL  label          Br Rel Long on Not Zero       RIL       BRCL  7,label
BREL   label          Br Rel Long on Equal          RIL       BRCL  8,label
BRZL   label          Br Rel Long on Zero           RIL       BRCL  8,label
BRNLL  label          Br Rel Long on Not Low        RIL       BRCL  11,label
BRNML  label          Br Rel Long on Not Minus      RIL       BRCL  11,label
BRNHL  label          Br Rel Long on Not High       RIL       BRCL  13,label
BRNPL  label          Br Rel Long on Not Plus       RIL       BRCL  13,label
BRNOL  label          Br Rel Long on Not Overflow   RIL       BRCL  14,label
BRUL   label          Unconditional Br Rel Long     RIL       BRCL  15,label
```

Jump on Condition Long

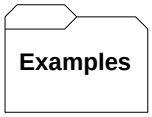
JLNOP	label	No operation	RIL	BRCL 0,label
JLO	label	Jump Long on Overflow	RIL	BRCL 1,label
JLH	label	Jump Long on High	RIL	BRCL 2,label
JLP	label	Jump Long on Plus	RIL	BRCL 2,label
JLL	label	Jump Long on Low	RIL	BRCL 4,label
JLM	label	Jump Long on Minus	RIL	BRCL 4,label
JLNE	label	Jump Long on Not Equal	RIL	BRCL 7,label
JLNZ	label	Jump Long on Not Zero	RIL	BRCL 7,label
JLE	label	Jump Long on Equal	RIL	BRCL 8,label
JLZ	label	Jump Long on Zero	RIL	BRCL 8,label
JLNL	label	Jump Long on Not Low	RIL	BRCL 11,label
JLNM	label	Jump Long on Not Minus	RIL	BRCL 11,label
JLNH	label	Jump Long on Not High	RIL	BRCL 13,label
JLNP	label	Jump Long on Not Plus	RIL	BRCL 13,label
JLNO	label	Jump Long on Not Overflow	RIL	BRCL 14,label
JLU	label	Unconditional Jump Long	RIL	BRCL 15,label

Tips

1) BRCL and the extended mnemonics that generate BRCL's represent a distinct improvement over BC's. Branch instructions specify their target addresses using base/displacement format, so for long programs, several base registers may be needed to cover all the target addresses. With relative branches, the target address is specified as a number of halfwords from the current instruction. No base registers are needed for target addresses.

2) Use jump (J) mnemonics instead of branch relative (BR) mnemonics. Jumping is an accurate description of how these instructions operate.

3) Abandon BC's for any new code you develop – choose jumps. You can also easily replace many older branch instructions with jumps as a way to reclaim the use of a registers that were given over to base/displacement addressing. Registers are always at a premium, and saving a register by converting your code to use jumps is a no-brainer!



Some Unrelated Branch Relative on Conditions

```
       LTR  R8,R8        SET THE CONDITION CODE
       JP  HERE          JUMP IF CONDITION CODE IS POSITIVE
       ...               OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
HERE   EQU *
       CLC  X,Y          SET THE CONDITION CODE
       JE     THERE      JUMP IF X = Y
       ...               OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
THERE  EQU
*
       CLC  X,Y          SET THE CONDITION CODE
       BRE  YON          JUMP IF X = Y  (Equivalent to JE)
       ...               OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
YON    EQU *
```

Trying It Out in VisibleZ:

1) Load the program **brcl.obj** from the \Codes directory. The first instruction is BASR which sets up addressability in R12. The second instruction, CR, sets the condition code to Equal/Zero. This is followed by a load which initializes R5 with a binary fullword. Finally we encounter a BRCL. What is the mask? What is the condition code? Why is byte 14 light red? The I2 value is 6. Since the mask was zero, a branch is not taken. The next instruction is another BRCL. Why is byte 0 light red?

2) Load the program **brcl1.obj**. Why does the first **BRCL** fall through? Why does the second **BRCL** take the branch?