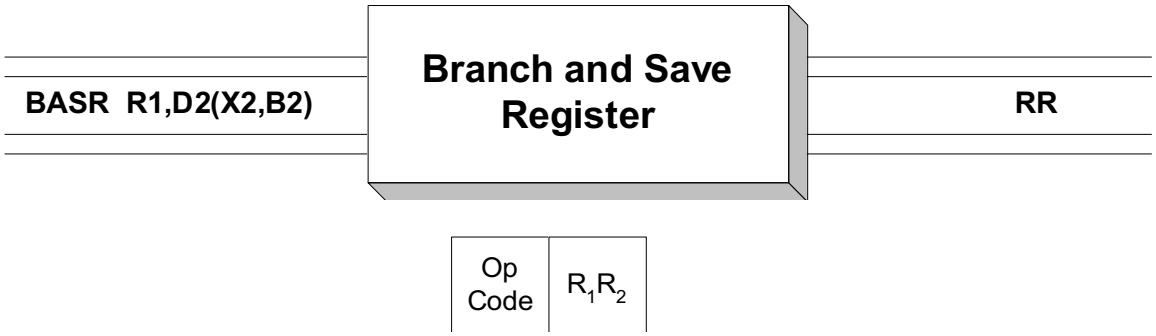
The BASR Instruction

A Visible/Z Lesson



The Idea:

BASR is a RR instruction which is used to support linkage and internal subroutines. When executed, the address of the instruction which follows the **BASR**, a return address, is stored in the operand 1 register, and a branch is taken to the address in the operand 2 register. No branch is taken if R0 is specified as operand 2. In this case, execution continues with the instruction following the **BASR**. Consider the instruction sequence below

```
          LA    R15,SUB1  PUT TARGET ADDRESS IN R15
          BASR  R14,R15   SAVE "HERE" AS THE RETURN ADDRESS
HERE      EQU   *         THE RETURN ADDRESS
          MVC   X,Y
          ...
SUB1      EQU   *
          ...   (SUBROUTINE CODE GOES HERE)
          BR    R14     BRANCH TO THE RETURN ADDRESS
```

First the address of the subroutine (the target address) is loaded into register 15. When the **BASR** is executed, the address of the next instruction (**MVC**) is loaded into R14. Recall that the EQU directive does not generate object code, and so the address denoted by HERE is equivalent to the address of the **MVC** instruction. After the return address is loaded, a branch occurs to the address in R15 (SUB1). This begins execution of the code in the subroutine. At completion of the subroutine, an unconditional branch occurs to the address in R14. This causes execution to resume at the **MVC** instruction. Care should be taken not to fall into the SUB1 code, otherwise the return address in R14 will be incorrect.

BASR can also be used in the format listed below.

```
          BASR    R12,R0
```

In this case, the address of the next instruction is loaded into R12, and no branch is taken since operand 2 was specified as R0. This form of a **BASR** is used in the topic called **Base / Displacement Addressing**. Briefly, R12 is declared to be a base register in a **USING** statement and the **BASR** loads R12 with the base address (the address of the next instruction).

BASR replaces an older instruction called "**BALR**". Both of these instructions load the address of the next instruction into operand 1. The difference in their operation depends on the addressing mode that the machine is using:

In 24 bit mode: BALR loads bits 0 - 7 of operand 1 with linkage information (instruction length code, condition code, program mask)
BALR loads bits 8 - 31 of operand 1 with the 24-bit return address

BASR loads bits 0 - 7 with eight 0's
BASR loads bits 8 - 31 of operand 1 with a 24-bit return address

In 31 bit mode BASR and BALR load bit 0 with a 1 indicating 31-bit mode addressing
BASR and BALR load bits 1 - 31 with a 31 bit-return address

The information that was provided by BALR in bits 0 - 7, can now be obtained using the **IPM** (Insert Program Mask) instruction.

Some Unrelated BASR's

```
BASR    R4,R5  LOAD NEXT ADDRESS IN R4, BRANCH TO ADDRESS IN R5
BASR    R4,R0  LOAD NEXT ADDRESS IN R4, DON'T BRANCH
BASR    R3,R0  LOAD NEXT ADDRESS IN R3, DON'T BRANCH
```

Tips

1) Use **BASR** instead of **BALR** to avoid non-zero bits being placed in the high-order byte of the stored address.

2) Read about the use of **BASR** in the topic **Base Displacement Addressing**.

Trying It Out in VisibleZ:

1) Load the program **basr.obj** from the \Codes directory and single step through each instruction.

2) The program loads a series of addresses into R12. Why is each address two larger than the previous address?

3) Does each BASR cause a branch? Why or why not?

4) Load the program **basr1.obj** from the \Codes directory and single step through each instruction.

5) How is the program able to return the second instruction?