Chapter 7: Slowly I Turn, …

In which we learn how to go from here to there and back again.

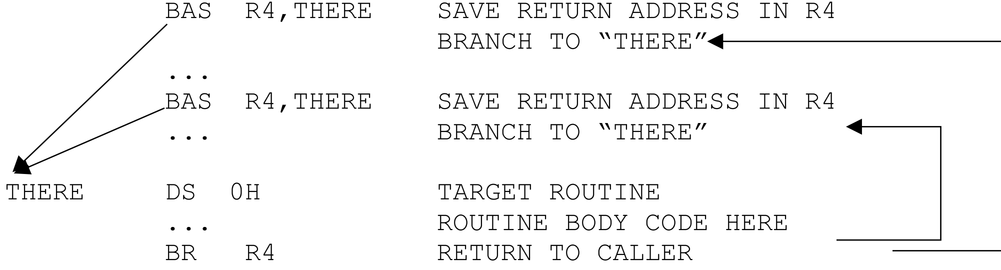
Internal Subroutines

Assembly language offers several alternatives for organizing a program into blocks of code that can be reused. Well-designed programs depend on this principle. In this chapter we consider **internal subroutines**. An **internal subroutine** consists of code that resides in the same control section as the code that is calling it. Later, we will tackle **external subroutines** which can reside in separately assembled modules and then linked together with the calling code or called dynamically. The techniques for calling internal and external subroutines differ considerably owing to differences in where the modules reside.

Conceptually, internal subroutines are much simpler to create than external ones. We only need to invoke any one of these three instructions: **BAS**, **BRAS**, or **BRASL**, to branch to a subroutine. When we are finished in the subroutine, a **BR** instruction is used carry us back to the return address. **BAS** stands for "branch and save". **BRAS** stands for "branch relative and save". **BRASL** stands for "branch relative and save long". Each of these might better be named "save and branch", because the first thing that happens is the address of the next instruction (the one following the **BAS** or **BRAS** or **BRASL**) is saved in the operand 1 register. This address will act as a return address. The only difference in these instructions is the technique for representing the target address. With **BAS** the target address is represented using a base/displacement address. With **BRAS** and **BRASL** the target address is represented as the number of halfwords from the beginning byte of the instruction. With **BRAS**, the number of halfwords is represented using 16 bits, and with **BRASL** 32 bits are used. With **BRAS**, the branch can cover roughly 64k bytes forward or backward from the first byte of the instruction. **BRASL** is available for even longer branches.

Today's machines support trimodal addressing with 24, 31, and 64-bit addresses. It is possible to call a subroutine that uses a different addressing mode from the caller. A different collection of branch instructions is needed for these types of calls and is beyond the scope of this book. The three branch and save instructions mentioned above essentially replaced an older instruction call **BAL**, Branch and Link, which was used for programs written in 24-bit addressing mode.

After the target routine has executed, a branch is taken to the address specified by operand 2 using **BR** (Branch Unconditionally Register) which provides an unconditional branch to the return address stored in operand 1 – a register. Here is an example of how these two commands can be used to create a call and return.

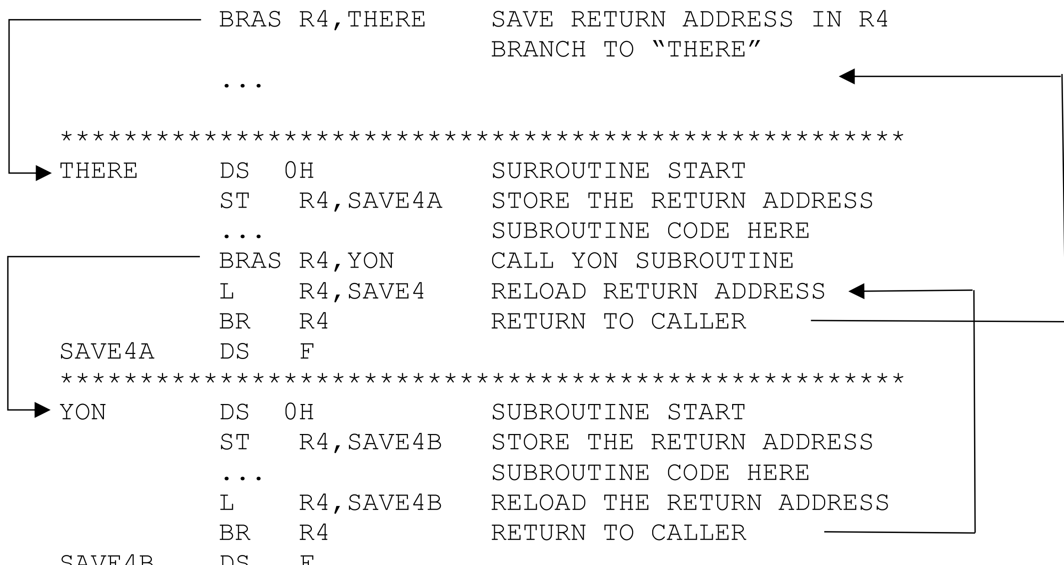


The first BAS branches to the subroutine after saving the return address (the next instruction address) in register 4. The subroutine executes and control is returned at the first return address when BR is executed. This acts as an unconditional branch to the address in R4. The process repeats with the second BAS after it saves a different return address.

Subroutines can call other subroutines using the same technique and even the same register, but they will need to follow a protocol:

1. Create a fullword inside the subroutine used to store the return address,
2. Store the contents of the register which was used to save the return address (let's call it a linkage register) upon entry to the routine in this fullword, and
3. Restore the linkage register contents from the fullword before exiting the routine with BR.

We will cover the required instructions (ST - Store Fullword and L - Load Fullword) more fully in a later chapter, but here's a simple example for the impatient or curious. Briefly stated, Store Fullword (ST) copies the contents of a register into a fullword in memory. Load Fullword (L) copies a fullword from memory into a register.



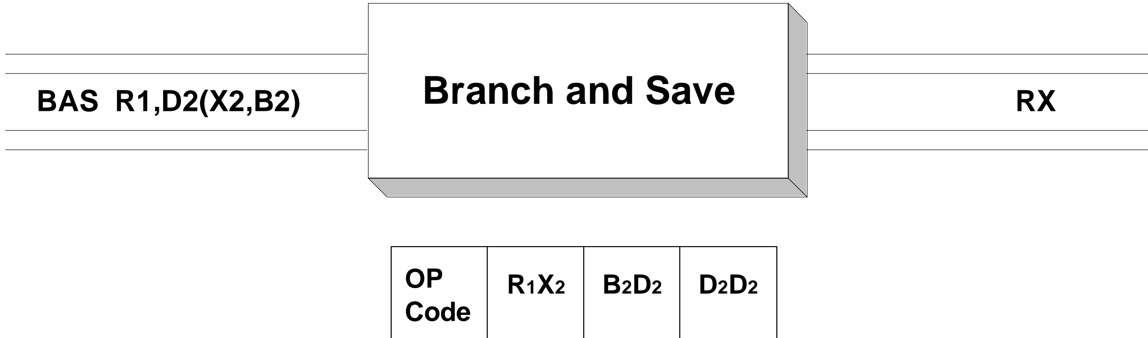
```
          BRAS R4,THERE    SAVE RETURN ADDRESS IN R4
                           BRANCH TO "THERE"
          ...

*****************************************************
THERE     DS   0H          SURROUTINE START
          ST   R4,SAVE4A   STORE THE RETURN ADDRESS
          ...              SUBROUTINE CODE HERE
          BRAS R4,YON      CALL YON SUBROUTINE
          L    R4,SAVE4    RELOAD RETURN ADDRESS
          BR   R4          RETURN TO CALLER
SAVE4A    DS   F
*****************************************************
YON       DS   0H          SUBROUTINE START
          ST   R4,SAVE4B   STORE THE RETURN ADDRESS
          ...              SUBROUTINE CODE HERE
          L    R4,SAVE4B   RELOAD THE RETURN ADDRESS
          BR   R4          RETURN TO CALLER
SAVE4B    DS   F
```

Upon entering the routine, we store the return address in SAVE4. Upon returning from the YON routine, we restore the return address for THERE by loading SAVE4. If all routines in a calling sequence follow the protocol, a single linkage register will support the flow of control for the entire sequence.

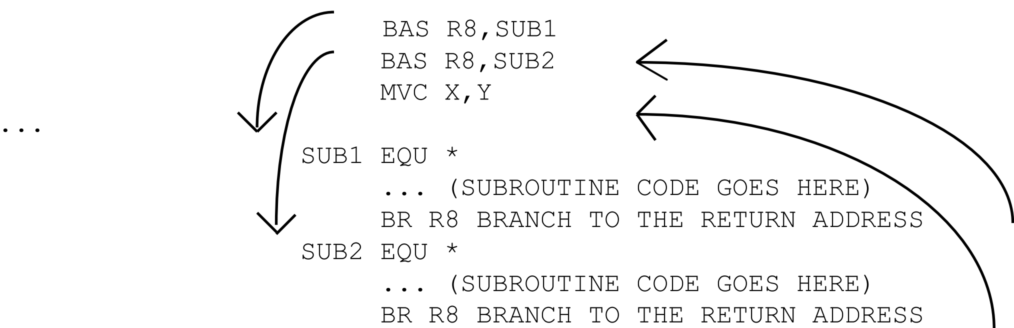
Finally, always use an unconditional "branch register" instruction, **BR**, to return to the calling routine. **BR** is an extended mnemonic instruction for the native **BCR** (Branch on Condition Register) which itself is the register version of another native instruction **BC** (Branch on Condition) that we covered in the chapter 5.

To reiterate, **BC** and **BCR** are native instructions, and you can read about them in Principles of Operation. There are numerous extended mnemonic instructions related to these instructions which make them easier to use. **BR**, the instruction we are using in this chapter, is one of them. You can read about extended mnemonics in the HLASM Language Reference.



BAS is a RX instruction which is used to support internal subroutines. When executed, the address of the instruction which follows the **BAS**, a return address, is stored in the operand 1 register, and a branch is taken to the address specified by operand 2. **BAS** is used in combination with **BCR** (technically, **BR**, an extended mnemonic of **BCR**) to construct internal subroutines (routines that are contained in the same control section).

Consider the instruction sequence below



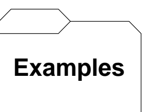
When the first **BAS** instruction is executed, the address of the next instruction (`BAS R8,SUB2`) is loaded into R8. After this return address is loaded, a branch occurs to the address denoted by SUB1. This begins execution of the code in the subroutine. At completion of the subroutine, an unconditional branch (`BR R8`) returns control to the address in R8. Execution resumes at the second **BAS** instruction. **BR** is an extended mnemonic of **BCR**, and branches unconditionally to the address in the operand 1 register. The second **BAS** causes the address of the next instruction (`MVC X,Y`) to be loaded into R8. A branch is taken to the subroutine denoted by `SUB2`. After execution of the subroutine, the unconditional branch (`BR R8`) at the end of the subroutine returns control at the **MVC** instruction.

BAS replaces an older instruction called "**BAL**" which stands for "Branch and Link". 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: BAL loads bits 0 - 7 of operand 1 with linkage information (instruction length code, condition code, program mask)
- BAL loads bits 8 - 31 of operand 1 with the 24-bit return address
- BAS loads bits 0 - 7 with eight 0's
- BAS loads bits 8 - 31 of operand 1 with a 24-bit return address

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

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



Some Unrelated BAS's

```
BAS  R4,HERE      LOAD NEXT ADDRESS IN R4, BRANCH TO "HERE"
BAS  R8,THERE     LOAD NEXT ADDRESS IN R8, BRANCH TO "THERE"
BAS  R10,YON      LOAD NEXT ADDRESS IN R10, BRANCH TO "YON"
```

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

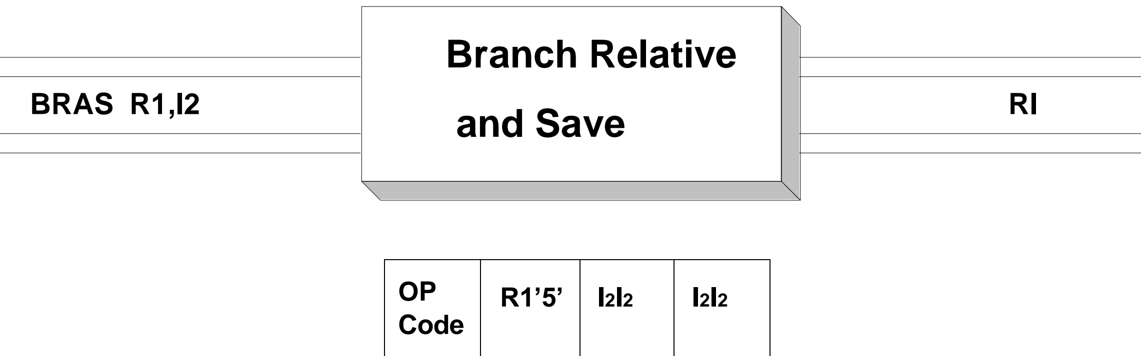
2. When creating internal subroutines, consider saving the linkage register on entry to the subroutine and restoring it just before exiting:

```
SUB1  EQU  *
      ST  R8,SAVE8  SAVE THE RETURN ADDRESS LOCALLY
      ...  (SUBROUTINE CODE GOES HERE)
      L  R8,SAVE8  RESTORE THE RETURN ADDRESS
      BR  R8
      DS  0F
SAVE8 DS  F
```

   By saving and restoring the linkage register, the subroutine is free to call other internal subroutines with the same linkage register.

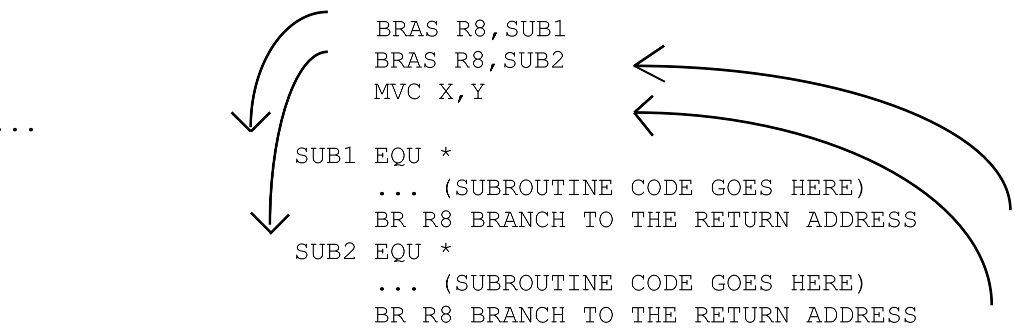
Trying It Out in VisibleZ:

1. Load the program **bas.obj** from the \Codes directory and single step through each instruction until you are about to execute the BAS instruction.
2. Which register will be used to hold the return address?
3. What is the return address?
4. What is the index register?
5. Where is the target of the address?
6. Cycle through the BAS instruction. What address was stored in R4? Why?
7. Cycle instructions until you are at BCR. What is the mask for this BCR? What are the conditions under which the branch will occur?
8. We have executed a "subroutine" and now we are returning. Is the return target address correct?



BRAS is a RI instruction which is used to support internal subroutines. When executed, the address of the instruction which follows the **BRAS**, a return address, is stored in the operand 1 register, and a branch is taken to the address specified by operand 2. **BRAS** is used in combination with **BCR** (technically, **BR**, an extended mnemonic of **BCR**) to construct internal subroutines (routines that are contained in the same control section).

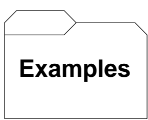
Consider the instruction sequence below



When the first `BRAS` instruction is executed, the address of the next instruction (`BRAS R8,SUB2`) is loaded into R8. After this return address is loaded, a branch occurs to the address denoted by `SUB1`. This begins execution of the code in the subroutine. At completion of the subroutine, an unconditional branch (`BR R8`) returns control to the address in R8. Execution resumes at the second `BRAS` instruction. **BR** is an extended mnemonic of **BCR**, and branches unconditionally to the address in the operand 1 register. The second `BRAS` causes the address of the next instruction (`MVC X,Y`) to be loaded into R8. A branch is taken to the subroutine denoted by `SUB2`. After execution of the subroutine, the unconditional branch (`BR R8`) at the end of the subroutine returns control at the `MVC` instruction.

The main difference between **BAS** and **BRAS** is in the technique used to denote the target address. With **BAS**, a base/displacement address is generated. With **BRAS**, a 16-bit, two's complement integer, I2, represents the number of halfwords to the target, measured from the first byte of the **BRAS** instruction. As a result, the target address can be forward ($2^{15} – 1$) bytes or backward 2^{15} bytes.

BRAS can be used instead of **BAS** to provide register relief since BRAS uses relative addressing.



Some Unrelated BRAS's

```
BRAS  R4,HERE      LOAD NEXT ADDRESS IN R4, BRANCH TO "HERE"
BRAS  R8,THERE     LOAD NEXT ADDRESS IN R8, BRANCH TO "THERE"
BRAS  R10,YON      LOAD NEXT ADDRESS IN R10, BRANCH TO "YON"
```

Tips

1. Use **BRAS** instead of **BAS** to reduce your dependence on base registers.

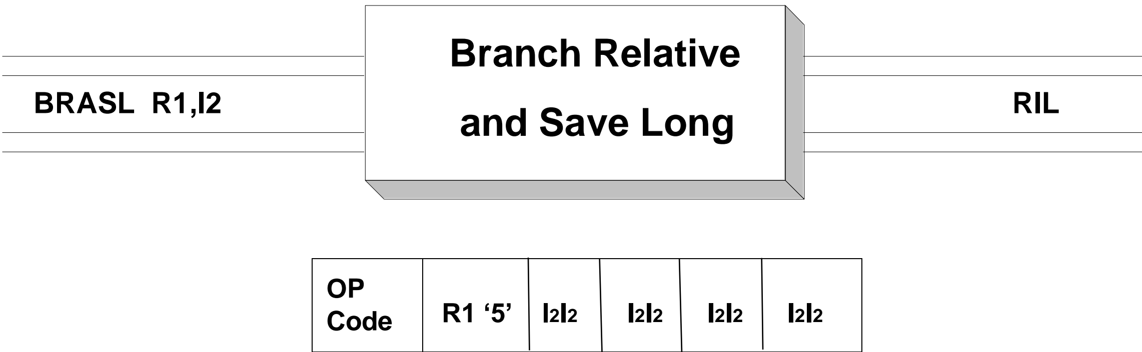
2. When creating internal subroutines, consider saving the linkage register on entry to the subroutine and restoring it just before exiting:

```
SUB1  EQU  *
      ST  R8,SAVE8  SAVE THE RETURN ADDRESS LOCALLY
      ...  (SUBROUTINE CODE GOES HERE)
      L  R8,SAVE8  RESTORE THE RETURN ADDRESS
      BR  R8
      DS  0F
SAVE8 DS  F
```

By saving and restoring the linkage register, the subroutine is free to call other internal subroutines with the same linkage register.

Trying It Out in VisibleZ:

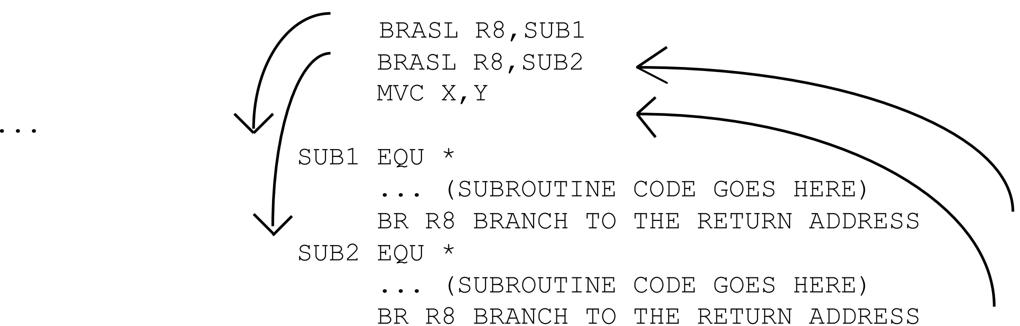
1. Load the program **bras.obj** from the \Codes directory and single step through each instruction until you are about to execute the BAS instruction.
2. Which register will be used to hold the return address?
3. What is the return address?
4. What is the I2 two's complement integer?
5. Where is the target of the address?
6. Cycle through the BRAS instruction. What address was stored in R4?
7. Cycle instructions until you are at BCR. What is the mask for this BCR? What are the conditions under which the branch will occur?
8. We have executed a "subroutine" and now we are returning. Is the return target address correct?



BRASL is a RIL instruction which is used to support internal subroutines. When executed, the address of the instruction which follows the **BRASL**, a return address, is stored in the operand 1 register, and a branch is taken to the address specified by operand 2. **BRASL** is used in combination with **BCR** (technically, **BR**, an extended mnemonic of **BCR**) to construct internal subroutines (routines that are contained in the same control section).

Consider the instruction sequence below

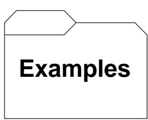
...



When the first `BRASL` instruction is executed, the address of the next instruction (`BRASL R8,SUB2`) is loaded into R8. After this return address is loaded, a branch occurs to the address denoted by `SUB1`. This begins execution of the code in the subroutine. At completion of the subroutine, an unconditional branch (`BR R8`) returns control to the address in R8. Execution resumes at the second `BRASL` instruction. **BR** is an extended mnemonic of **BCR**, and branches unconditionally to the address in the operand 1 register. The second `BRASL` causes the address of the next instruction (`MVC X,Y`) to be loaded into R8. A branch is taken to the subroutine denoted by `SUB2`. After execution of the subroutine, the unconditional branch (`BR R8`) at the end of the subroutine returns control at the `MVC` instruction.

The main difference between **BAS** and **BRASL** is in the technique used to denote the target address. With **BAS**, a base/displacement address is generated. With **BRASL**, a 32-bit, two's complement integer, I2, represents the number of halfwords to the target, measured from the first byte of the BRASL instruction. As a result, the target address can be forward $(2^{31} - 1)$ bytes or backward 2^{31} bytes.

BRASL can be used instead of **BAS** to provide register relief since BRASL uses relative addressing.



Some Unrelated BRASL's

```
BRASL  R4,HERE      LOAD NEXT ADDRESS IN R4, BRANCH TO "HERE"
BRASL  R8,THERE     LOAD NEXT ADDRESS IN R8, BRANCH TO "THERE"
BRASL  R10,YON      LOAD NEXT ADDRESS IN R10, BRANCH TO "YON"
```

Tips

1. Use **BRASL** instead of **BAS** to reduce your dependence on base registers.

2. When creating internal subroutines, consider saving the linkage register on entry to the subroutine and restoring it just before exiting:

```
SUB1  EQU  *
      ST  R8,SAVE8  SAVE THE RETURN ADDRESS LOCALLY
      ...  (SUBROUTINE CODE GOES HERE)
      L  R8,SAVE8  RESTORE THE RETURN ADDRESS
      BR  R8
      DS  0F
SAVE8 DS  F
```

   By saving and restoring the linkage register, the subroutine is free to call other internal subroutines with the same linkage register.

Trying It Out in VisibleZ:

1. Load the program **brasl.obj** from the \Codes directory and single step through each instruction until you are about to execute the BAS instruction.
2. Which register will be used to hold the return address?
3. What is the return address?
4. What is the I2 two's complement integer
5. Where is the target of the address?
6. Cycle through the BRAS instruction. What address was stored in R4?
7. Cycle instructions until you are at BCR. What is the mask for this BCR? What are the conditions under which the branch will occur?
8. We have executed a "subroutine" and now we are returning. Is the return target address correct?

Branch On Condition Register

BCR M1,R2 **RR**

OP Code	M_1R_2

The Branch on Condition Register 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 (binary pattern) indicating the conditions under which the branch should occur. Operand 2 is a register containing the target address to which the branch will be made if the condition indicated in Operand 1 occurs. If the condition code is one of the values specified in the mask, the instruction address in the PSW is replaced with the target address in R_2. This causes the processor to fetch the instruction located at the target address as the next instruction in the fetch/execute cycle.

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 instructions,

```
LA   3,THERE
BCR  8,3
```

In the BCR, 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 BCR above.

```
BCR   B'1000',3
```

Extended mnemonics were developed to replace the awkward construction of having to code a mask. The extended mnemonics are easier to code and read. A listing of the extended mnemonics follows below.

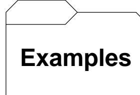
BER	Branch Equal Register	BNER	Branch Not Equal Register
BZR	Branch Zero Register	BNZR	Branch Not Zero Register
BLR	Branch Low Register	BNLR	Branch Not Low Register
BMR	Branch Minus Register	BNMR	Branch Not Minus Register
BHR	Branch High Register	BNHR	Branch Not High Register
BPR	Branch Positive Register	BNPR	Branch Not Positive Register
NOPR	No Operation Register	BR	Unconditional Branch Register

Using extended mnemonics, we could replace the previous Branch On Condition Register instruction with the following,

```
BZR   3
```

The "BZR" means "Branch on Condition Zero Register". When the assembler processes this **BZR**, it generates the mask as B'1000'. The table below indicates the possible mask values and the equivalent extended mnemonics. Notice that not all masks have an equivalent extended mnemonic.

Eq/Low	Low/Min	High/Plus	Overflow	Decimal Condition	Extended Mnemonic
0	0	0	0	0	NOPR
0	0	0	1	1	BOR
0	0	1	0	2	BHR, BPR
0	0	1	1	3	No mnemonic
0	1	0	0	4	BLR, BMR
0	1	0	1	5	No mnemonic
0	1	1	0	6	No mnemonic
0	1	1	1	7	BNER, BNZR
1	0	0	0	8	BER, BZR
1	0	0	1	9	No mnemonic
1	0	1	0	10	No mnemonic
1	0	1	1	11	BNLR, BNMR
1	1	0	0	12	No mnemonic
1	1	0	1	13	BNHR, BNPR
1	1	1	0	14	No mnemonic
1	1	1	1	15	BR



Some Unrelated Branch on Condition Register Examples

```
         LA    7,THERE   POINT REGISTER AT TARGET ADDRESS
         LTR   R8,R8     SET THE CONDITION CODE
         BPR   7         BRANCH IF CONDITION CODE IS POSITIVE
         ...             OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
THERE    EQU   *

         LA    5,THERE   POINT REGISTER AT TARGET ADDRESS
         CLC   X,Y       SET THE CONDITION CODE
         BER   5         BRANCH IF X = Y
         ...             OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
THERE    EQU   *


         LA    9,THERE   POINT REGISTER AT TARGET ADDRESS
         CLC   X,Y       SET THE CONDITION CODE
         BHR   9         BRANCH IF X > Y
         ...             OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
THERE    EQU   *

         LA    R6,THERE  LOAD R6 WITH THE TARGET ADDRESS
         BR    R6        BRACH TO THE ADDRESS CONTAINED IN R6

         THIS LAST EXAMPLE IS FUNCTIONALLY EQUIVALENT TO THIS EXAMPLE
         WHICH DOESN'T USE A REGISTER EXPLICITLY:

         B     THERE
```

PROBLEMS

7-1. When BAS, BRAS, or BRASL are executed, what is being saved? Where is it saved?
7-2. How is the target address stored in a BAS instruction?
7-3. How is the target address stored in a BRAS instruction?
7-4. Can an internal routine use BRAS to call itself? What problem will result?
7-5. Why is it a usually a good idea for an internal routine to save and restore the calling register?

Programming Exercise

1. Write a main program that uses two internal subroutines described below to print a tic-tac-toe diagram:

```
       *     *
       *     *
       *     *
*********************
       *     *
       *     *
       *     *
*********************
       *     *
       *     *
       *     *
```

Write a subroutine that prints a single line:

```
       *     *
```

Write a second subroutine that prints a single line:

```
*********************
```

2. Reusing your code from #1, write a third routine that invokes the first two routines to print two lines:

```
*********************
       *     *
```

Try writing the code so that it only uses a single register for all the branches. Modify your main program to print this:

```
*********************
       *     *
*********************
       *     *
*********************
       *     *
*********************
       *     *
*********************
```