LOOPS

A **loop** is a control structure which allows a block of instructions, the **loop body**, to be executed repeatedly in succession. In this article we investigate loop control structures and how they are constructed in assembly language. There are two categories of loops:

1) **Counted Loops** - Loops in which the iteration number, the number of times the loop should be executed, is known before the loop is entered.

2) **Conditional Loops** - Loops which will be continually iterated until a prescribed condition occurs.

Creating Counted Loops

The most commonly used instruction for creating a counted loop is the Branch On Count instruction, which has mnemonic **BCT**. The iteration number is first loaded into a register which is subsequently manipulated by **BCT**. Each time the **BCT** is executed, the iteration number in the register is decremented by one. After the register is decremented, the register is tested. If the result in the register is not zero, a branch is taken to the address specified in Operand 2. If the test reveals that the result in the register is zero, the loop is terminated and execution continues with the instruction following the **BCT**. Here is a sample loop that would be iterated 10 times.

```
LA      R5,10             PUT THE ITERATION NO. IN R5
LOOPTOP EQU   *

        ...LOOP BODY GOES HERE

        BCT   R5,LOOPTOP  DECREMENT R5,IF NOT 0, BRANCH BACK
```

In the code above, the register that will control the loop is initialized at 10 and then the statements in the loop body are executed. At the end of the first iteration, **BCT** subtracts 1 from register 5, leaving it set to 9. The register is tested and since it does not contain zero, a branch is taken back to LOOPTOP. The loop body is executed a second time, and again **BCT** subtracts 1 from register 5, leaving it set at 8. **BCT** tests the content of R5, which is now 8 , and not finding it zero, branches back to LOOPTOP. This process continues through 10 iterations of the loop body. On the tenth iteration, **BCT** decrements the register and the result becomes zero. Testing R5 reveals it to be zero. This time the branch is not taken and control returns to the statement following the branch on count instruction.

There are two other instructions which are occasionally used to implement a counted loop. The first instruction that we consider is called “Branch on Index Less Than or Equal”. The mnemonic for this instruction is **BXLE**. When coded it has three operands:

1) Operand 1 is a register containing a count or an address.

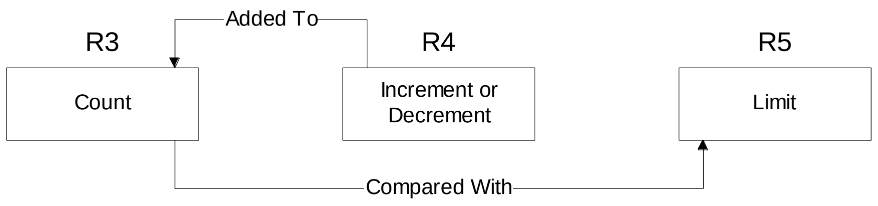
2) Operand 2 is typically an even register of an even-odd consecutive pair of registers. The even register contains a value which is used to increment or decrement the value in Operand 1.
The odd register, which is not coded in the instruction, contains a limit or address against which Operand 1 is compared.

3) Operand 3 represents an address to which the instruction will branch if the Operand 1 value is less than or equal to the limit in the odd register.

For example, consider the following instruction.

```
                    BXLE   R3,R4,LOOPTOP
```

The diagram below illustrates the relationships among the registers used by the instruction.



Each time the instruction is executed, the value in the even register, R4, is added to the value in the Operand 1 register, R3. If the result is less than or equal to the value in the odd register, R5, a branch occurs to the address in Operand 3, LOOPTOP. The code listed below uses **BXLE** to implement a loop.

```
          SR     R3,R3      PUT 0 IN R3
          LA     R4,10      PUT INCREMENT INTO EVEN REG
          LA     R5,100     PUT LIMIT VALUE IN ODD REG
LOOPTOP   EQU    *

                 ...LOOP BODY

                 BXLE  R3,R4,LOOPTOP   EXECUTE THE LOOP
```

First, the count in R3 is initialized to 0, the increment is set to 10, and the limit is set at 100. Each time the **BXLE** is executed, the increment in R4 is added to the count in R3 and the result is compared to the limit in R5. If the new count is less than or equal to the limit, a branch occurs back to LOOPTOP. The effect of the statement is to execute the loop 11 times. (The loop is executed once on the equal condition.)

BXLE has a companion instruction called “Branch on Index High”. The mnemonic is **BXH** and the instruction is similar in execution to **BXLE** except the branch occurs on a “high” condition instead of “less than or equal”. The following code gives an example of this instruction.

```
          LA     R3,5       SET THE COUNT TO 5
          L      R4,=F’-1’  SET THE DECREMENT TO -1
          SR     R5,R5      SET THE LIMIT VALUE AT 0
LOOPTOP   EQU    *

                 ...LOOP BODY

                 BXH  R3,R4,LOOPTOP   EXECUTE THE LOOP
```

In the example above, the count is set at 5, the decrement is -1, and the limit is 0. Each time the BXH is executed, the decrement is added to the count, reducing it by 1. As long as the result is higher than the limit 0, a branch occurs to LOOPTOP. The effect is to execute the loop body 5 times.

The main drawback to using **BXLE** and **BXH** is that both instructions require 3 registers. Since registers are usually at a premium in most programs, loops are often implemented using **BCT**.

While the instructions discussed above were specifically designed for the construction of loops, comparisons of any type, as well as other instructions that set the condition code, can be used to create "home-made" loop structures. In the code below we use a packed decimal instruction in combination with a branch to implement a counted loop.

```
         ZAP   COUNT,=P'100'     LOOP 100 TIMES
LOOPTOP  EQU   *

         ...LOOP BODY

         SP    COUNT,=P'1'       DECREMENT ON EACH ITERATION
         BNZ   LOOPTOP           BRANCH IF NOT ZERO
```

In this case, we have decided to loop 100 times. Each time through the loop the count is decremented by 1. We take advantage of the fact that **SP** sets the condition code based on the result of the subtraction. If the result is not zero, we loop back to LOOPTOP.

Creating Conditional Loops

Conditional loops are characterized by the property that we cannot predetermine the number of times the loop will be iterated. In other words, the loop will continue to be executed until a prescribed condition occurs. The condition is tested "by hand" using one of the compare instructions; **CP** for packed data, **CLC** or **CLI** for character data, and **C** or **CH** for binary data. The following code implements a conditional loop that continues to be executed until a field, called "FLAG", is equal to "Y".

```
LOOPTOP  EQU   *

         ...LOOP BODY

         CLI   FLAG,C'Y'     IS THE FLAG SET?
         BNE   LOOPTOP       NO... BRANCH BACK
```

When creating loops of this type, the loop body must contain logic that will eventually cause termination of the loop. Otherwise an infinite loop results.

One common use of a conditional loop is for processing all the records in a sequential file. This is illustrated with the code below.

```
GETREC   EQU   *
         GET   MYFILE,MYRECORD      READ A RECORD

         ...PROCESS THE RECORD

         B     GETREC               LOOP BACK FOR NEXT RECORD
NEXTSTEP EQU   *
```

At first glance, the loop above appears to be an "infinite" loop. In other words, there does not appear to be logic present that would allow the program to escape the loop body once it is entered. This problem is resolved when we execute the GET macro and there are no more records in the file. At this point, a branch would occur to NEXTSTEP if we have specified

EODAD=NEXTSTEP in the DCB of MYFILE. The EODAD parameter causes an unconditional branch to the address we specify in the parameter when “end of file” is detected. So, in fact, the loop above is conditional, and continues to execute until “end of file” occurs.

Occasionally you may code an infinite loop by mistake. When this happens, your program will continue to execute the loop until it has used up the time allocated to the job by the operating system. At that point, the program will be interrupted with a “322” abend.