The BR Instruction

The High Level
The **Branch** instruction is a type of **Program Control Instruction** that changes the order in which you execute instructions.

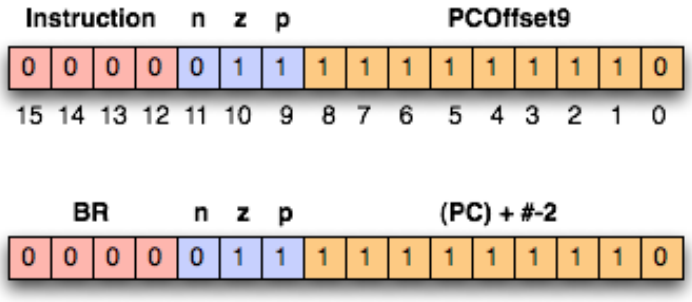

Figure 1: The BR Instruction (details)

Prerequisite Knowledge: To fully grasp the content of this document, you should thoroughly understand:

- What a PCOffset9 is
- The LD instruction
- The ADD instruction

The Details
This instruction will pop up almost every program you write. The BR instruction lets you hop around in your code rather than executing instructions {1, 2, 3, ..., n} in order. This is useful (read: necessary) when you want to implement a **For Loop**, **While Loop**, **Do Loop**, or an **If Statement**.

The BR instruction takes up to three parameters: **n**, **z**, and **p**. Whenever a register is **modified**, special flags in memory are updated with whether the register is now **n**egative, **z**ero, or **p**ositive. When you use a BR instruction, you can optionally add any combination of {**n**, **z**, **p**} after the "BR". Note, however, that the **n**, **z**, and **p** must appear in **n-z-p order** (i.e. BRpz or BRzn or BRpn are all **incorrect**).

The BR instruction, like LD and ST, makes use of a **PCOffset9**, which is denoted by a **label**. For a less "wordy" explanation, see Figure 2 and examples below.

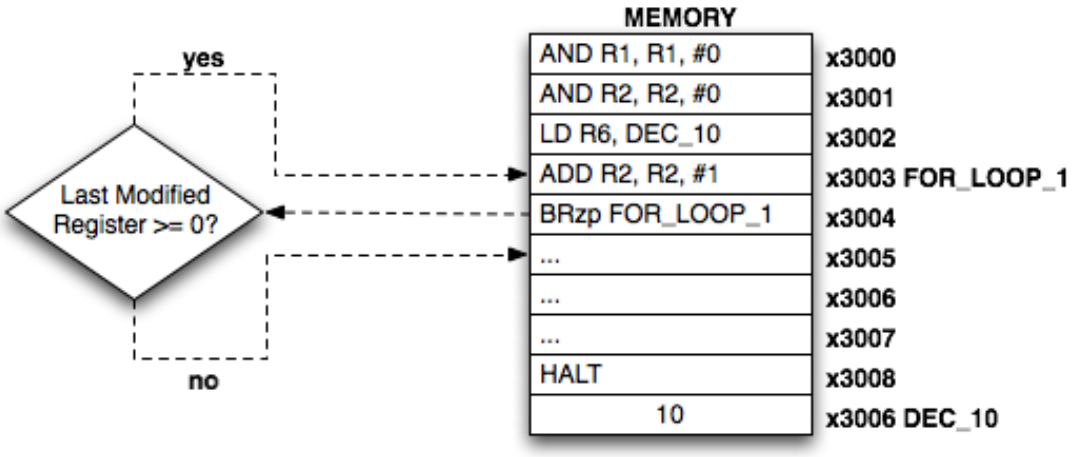

Figure 2: The BR Instruction – Visual Execution

The Breakdown:

- Bits [15-12] specify what instruction to execute (0000 is BR)
- Bit [11] denotes a condition: 1 → Last Modified Register must be negative
- Bit [10] denotes a condition: 1 → The Last Modified Register must be zero
- Bit [9] denotes a condition: 1 → The Last Modified Register must be positive
- Bits [8-0] specify a 9-bit PCOffset, which is the number of memory locations (i.e. "spaces") between the PC and the location you want to jump to

The Examples!

```
;-------------------------------------------------------------------
; Example of the following pseudocode code, translated into LC3:
;    if (R1 < 0)
;    {
;       goto LMR_IS_N
;    }
;-------------------------------------------------------------------
.orig x3000
;---------------
; Instructions
;---------------
   LD R1, DEC_NEG_10     ; R1 <-- #-10
   BRn LMR_IS_NEGATIVE   ; if (R1 < 0) { goto LMR_IS_N } else { continue }

LMR_IS_ZP
   LEA R0, MSG_1         ; R0 <-- Address of the beginning of the MSG_1 string
   PUTS                  ; Print the MSG_1 string
   HALT                  ; Stop executing program

LMR_IS_N
   LEA R0, MSG_1         ; R0 <-- Address of the beginning of the MSG_2 string
   PUTS                  ; Print the MSG_1 string
   HALT                  ; Stop executing program

;-------
; Data
;-------
DEC_NEG_10     .FILL     #-10
MSG_1          .STRINGZ  "LMR is zero or positive!"
MSG_2          .STRINGZ  "LMR is negative!"

.end                     ; No more assembly code to read into memory
```

```
;-------------------------------------------------------------------
; Example of the following pseudocode code, translated into LC3:
;    R1 = 0
;    for(i = 10; i > 0; i--)
;    {
;       R1 = R1 + 5
;    }
;-------------------------------------------------------------------
.orig x3000
;---------------
; Instructions
;---------------
   AND R1, R1, #0      ; R1 <-- 0
   LD R6, DEC_10       ; R6 <-- #10
   BRp FOR_LOOP_1      ; if (R6 >= 0) { goto FOR_LOOP_1 } else { continue }
   HALT                ; Terminate program

   FOR_LOOP_1
      ADD R1, R1, #5   ; R1 <-- R1 + 5
      ADD R6, R6, #-1  ; R6 <-- R6 - 1
      BRp FOR_LOOP_1   ; if (R6 > 0) { goto FOR_LOOP_1 } else { continue }

   HALT                ; Stop executing program

;-------
; Data
;-------
DEC_6  .FILL     #6
MSG_1  .STRINGZ  "LMR is zero or positive!"
MSG_2  .STRINGZ  "LMR is negative!"

.end                    ; No more assembly code to read into memory
```

Pitfalls... (aka: Erroneous code makes baby monkeys cry)
The example below is erroneous. Please do NOT try to code this way!

```
BRn x4000  ; (ERROR: You must use a label,not a literal memory address)
BRn LabelThatIsReallyFarAway ; (ERROR: Overflows 9-bit PCOffset9 field)

BRzn FOR_LOOP_1   ; (ERROR: Must be in n-z-p order. Should have been BRnz)
BRpz FOR_LOOP_1   ; (ERROR: Must be in n-z-p order. Should have been BRzp)
BRpz FOR_LOOP_1   ; (ERROR: Must be in n-z-p order. Should have been BRzp)
BRpzn FOR_LOOP_1   ; (ERROR: Must be in n-z-p order. Should have been BRnzp)
```

The first example pitfall code above is incorrect because the BR instruction requires the use of a label, not a memory address as it's operand.

The second example pitfall is incorrect because it overflows the 9-bit PCOffset field (i.e. the place to jump was more than 256 instructions away so it cannot "reach" it with 9 bits of range). *See LD or ST tutorials for the full explanation.*

The rest of the pitfall examples are all the combinations of ways to mess up the **n-z-p order**. The **n** must always come before the **z**, which must always come before the **p**.