instruction, we give the symbolic address of the instruction to be jumped to. This makes the Assembly language program far easier to follow. The assembler decodes the symbolic address into an absolute address, calculates the displacement necessary to get to the address, and writes that displacement into the machine code instruction. The symbolic address is called a *label*, and it's analogous to a BASIC program line number.

Let's take a closer look at how labels are used. A label is an alphanumeric string written at the start of an Assembly language instruction. It is treated by the assembler program as a two-byte symbol standing for the address of the first byte of the instruction. Therefore, we can re-write the program given in this way:

ORG $5E00		
	6502	Z80
5E00	ADC #$34	ADC A,$34
5E02	BEQ EXIT	JR Z,EXIT
5E04	STA $5E20	LD ($5E20),A
5E07 EXIT	RTS	RET

The instruction at $5E02 can now be read as 'IF the value of the accumulator is zero THEN GOTO the address represented by the label EXIT'. This is an enormous improvement in readability over the previous version, and greatly decreases the chance of miscalculating the jump destination.

We can now use labels and the branch instructions to create a loop:

ORG $5E00		
	6502	Z80
5E00 START	ADC #$34	ADC A,$34
5E02	BNE START	JR NZ,START
5E04	STA $5E20	LD ($5E20),A
5E07 EXIT	RTS	RET

Notice here the use of the new label, START, as well as the new branch instructions: BNE, meaning 'Branch if the accumulator is Not Equal to zero'; and JR NZ, meaning 'Jump if the accumulator is Not equal to Zero'. Let's consider what effect this code will have. The program will first add $34 to the accumulator. If the result is not equal to zero then the program branches back to $5E00 — the address represented by the label START. $34 will again be added to the accumulator, and the result will decide whether another branch occurs. This 'loop' will go on and on until the branch condition is met. When the contents of the accumulator do equal zero following an ADC instruction, then the branch at $5E02 will not occur, and the instruction at $5E04 will be executed next.

This is exactly like an IF...THEN GOTO... loop in BASIC, except that it's difficult to see how the accumulator could ever become zero. After all, it is being increased by $34 every time the loop is executed! How will it ever add up to zero? The answer lies in the fact that the accumulator is only a single-byte register, and if the addition results in a two-byte number, then the carry flag of the processor status register will be set, and the accumulator will hold the lo-byte of the result. If the accumulator contains $CC, for example, then adding $34 will give the two-byte number $0100. The carry flag will be set, and the accumulator will hold the lo-byte of this result — $00. Thus, the contents of the accumulator would be zero, and the zero flag set as a result.

With this result in mind, we might re-write the program to use a different branch condition, incorporating the state of the carry flag rather than the state of the zero flag.

ORG $5E00		
	6502	Z80
5E00 START	ADC #$34	ADC A,$34
5E02	BCC START	JR NC,START
5E04	STA $5E20	LD ($5E20),A
5E07 EXIT	RTS	RET

In this version, the instruction at $5E02 reads 'if the carry flag is clear, branch to START'. As soon as the result of adding $34 to the accumulator is greater than $FF, then the carry flag will be set, and the branch back to the START address will not occur.

LOOP COUNTERS

It may seem that branching according to the current condition of either the carry flag or the zero flag is a rather limited facility, but it permits a wide range of decision making, as we shall shortly see. What is definitely lacking from our repertoire now is the ability to keep a *loop counter*. We might wish, for example, to count the number of times that a loop is performed before the exit condition occurs, or we might want to cause an exit from the loop after a given number of iterations. The first objective is easily achieved by employing a CPU index register to hold the counter, and an increment instruction to update the counter:

6502		
0000	ORG	$5DFD
5DFD	LDX	#$00
5DFF START	INX	
5E00	ADC	#$34
5E02	BCC	START
5E04	STX	$5E20
5E07 EXIT	RTS	

Z80		
0000	ORG	$5DFA
5DFA	LD	IX,$0000
5DFE START	INC	IX
5E00	ADC	A,$34
5E02	JR	NC,START
5E04	LD	($5E20),IX
5E08 EXIT	RET	

The new structure has forced several changes in the program. Firstly, the instructions inserted at the beginning of the program require a new ORG address. These instructions have much the same effects on both the 6502 and the Z80 processors, but their lengths are different, so the location addresses are no longer the same in both versions of the program.

Secondly, new versions of the load (LDX, LD IX) and store (STX, LD( ),IX) instructions have been used to place an initial value of $00 in the CPU