COUNTER INSTRUCTIONS

Loops and conditional branches are implemented in Assembly language by using the processor status register flags to test the condition of the accumulator, and the relative jump instructions to change the flow of control in the program. These structures and the indexed addressing mode combine in creating data tables.

Before we can begin to use the various CPU addressing modes (especially indexed addresses) to advantage, we must first be able to write a loop. Without this fundamental structure we are in much the same position as a BASIC programmer who knows about arrays, but is ignorant of the FOR...NEXT command. There are no automatic structures like FOR...NEXT in Assembly language (though there is a Z80 instruction that is very close to it), but we can construct loops of the IF...THEN GOTO... type. These require instructions that make decisions or express conditions, and effectively change the order in which instructions are obeyed in the program.

Decision making in Assembly language centres on the flags in the processor status register. These flags show the effects on the accumulator of the last instruction executed, and are sometimes called *condition flags*. All these flags can be used in decision making, but we will need to consider only two of them at present — the zero (Z) and the carry (C) flags.

The state of these flags can be used to decide whether the processor executes the next instruction in the program, or whether it jumps to another instruction elsewhere in the program. The decision to continue or to jump is arrived at by the processor's either changing or accepting the address contained in its program counter. This register always contains the address of the next machine code instruction to be obeyed. When the processor begins to execute an instruction, it loads the op-code of the instruction from the byte pointed to by the address in the program counter. The address in the register is incremented by the number of bytes in the instruction so that the program counter then points to the op-code of the next instruction. If the current instruction causes the program counter to point to an address elsewhere in the program, then a jump is effectively generated.

On the 6502, the instruction BEQ causes the program counter to be changed if the zero flag is set. BCS is the equivalent instruction if the carry flag is set. On the Z80, these instructions are JR Z and JR C respectively. These four op-codes are called *branch instructions* because they represent a branch-point in the flow of program control. Their operand is a single-byte number, which is added to the address in the program counter to produce a new address. Consider what happens when the following program is executed:

ORG $5E00			
6502		Z80	
5E00	ADC #$34	ADC	A,$34
5E02	BEQ $03	JR	Z,$03
5E04	STA $5E20	LD	($5E20),A
5E07	RTS	RET	

If the ADC instruction at $5E00 produces a zero result in the accumulator (which is unlikely but, as we'll see later, possible), then the BEQ and JR Z instructions at $5E02 will cause $03 to be added to the contents of the program counter. The next instruction to be executed, therefore, will be the return instruction at $5E07, causing the instruction at $5E04 to be skipped over.

At first sight, this may seem wrong. After all, if the instruction at $5E02 causes $03 to be added to the program counter, surely the address stored there will become $5E05? But it is important to remember that the program counter always points to the *next* instruction to be executed and not the instruction currently being obeyed. Thus, when the instruction at $5E02 begins execution, the program counter will contain the address $5E04 — the location of the next instruction. If $03 is added to $5E04 the result will be $5E07, the address of the following instruction.

It's worth remarking here that the processor is not capable of checking whether the addresses pointed to are correct. If we inadvertently change the displacement in the instruction to $02, then the program counter will be increased (if the accumulator contains zero) by $02, and the processor will consider $5E06 to be the address of the op-code of the next instruction. In our correct program, $5E06 contains the value $5E, which is the hi-byte of the operand of the instruction at $5E04. The processor, however, cannot evaluate whether it is the right instruction or not. As far as it is concerned, $5E is a valid op-code and it will proceed to execute it, taking the bytes following $5E06 as the operands of the instruction. The program will probably crash as a result. Miscalculating displacements like this is one of the commonest errors in machine code programming.

In Assembly language programming, however, calculating jump displacements need not be a problem because the assembler program can do it for us. Therefore, instead of supplying a hex displacement as the operand of the branch