Relative Jumps

Most of the branch instructions, such as BCS (meaning 'branch if carry flag is set'), JR NZ (meaning 'branch if the accumulator is non-zero'), act according to the condition of the processor status register, and use the relative jump mode in redirecting the flow of control through the program. The alternative is the absolute jump.

In the example, the BCS $01 instruction always causes a relative jump of one byte forward (when it causes a jump at all, that is; it's conditional on the state of the carry flag) no matter what the location address at which the machine code resides. Here, the BCS $01 instruction is always followed by the INX instruction, itself only a single-byte instruction; when the carry flag is set, therefore, BCS will cause the INX instruction to be skipped.

Absolute Jumps

In this example, the JP $65A2 instruction causes an unconditional jump whenever it is encountered. Its effect is to redirect program execution to the address which forms its operand — $65A2 here. No testing is done, and the location address of the instruction at the time of execution is not significant; program execution always continues from the specified address.

Both jump modes have advantages and disadvantages, but the most important criterion in choosing between a relative jump or an absolute jump is relocatability: it's quite common in Assembly language programming to write a routine and assemble it at one ORG address, then re-use it in the same form but with a different ORG value. If all the jumps in the routine are relative, then changing the location addresses of the instructions will not matter at all, and the program will flow smoothly along its intended paths; if any of the jumps is absolute, however, when the routine is assembled at a different ORG, the jumps will still send control to the specified address, which may now have no significance for the routine. Relative jumps are relocatable, absolute jumps are not.

lo-byte of IX as the loop counter. The instruction LD IX,$5E00 puts the base address, $5E00, into the IX register, so the lo-byte of IX will contain $00. The peculiar-looking instruction LD (IX+$22),A means 'add the address contained in IX to $22, and store the contents of the accumulator at the address thus formed'. Since IX is initalised to $5E00, and is subsequently incremented at every loop interation, the starting value of the accumulator will be stored at $5E22, the next value at $5E23, and so on. Meanwhile the lo-byte of IX records the number of loop iterations, and is finally stored at $5E20 when the loop terminates. The LD (IX-$22),A instruction here is in the absolute indirect indexed addressing mode, which is rather more complicated than the 6502 version but much more powerful.

We have now looked at the Assembly language loop and array structures in some detail. These are both extremely helpful machine code programming techniques. In the next instalment of the course, we'll put them both to work.

Exercises

There are many important, and possibly puzzling, points in this instalment, and only experience of using the new addressing modes and instructions will make you fully understand them.

Use the CHAMP assembler package to assemble and SAVE the various program fragments in this instalment. When you execute a fragment, use the ‹debug› mode to examine the memory locations that should be affected. It's a good idea always to initialise these locations with a recognisable constant — $FF, for instance — before execution, so that afterwards you can tell whether memory has been affected by the program. You can use the ‹debug› Alter command to do this, or even the ‹debug› Move command.

Remember, as always, that the location addresses given in the program are for example only, and that you must choose addresses suitable for your machine.

Loading And Saving CHAMP

For convenience and security you should copy CHAMP onto another tape, and then remove the write-protect tabs from the original and the copy. In the following instructions, the LOAD instructions refer to the CHAMP tape, and SAVE refers to the copy tape:

BBC Model B

1) LOAD"CHAMP"
2) SAVE"CHAMP" : RUN : Quit to BASIC
3) *SAVE"CHAMP M/C" 1000 ,4600

Commodore 64

1) LOAD"CHAMP"
2) SAVE"CHAMP" : RUN : enter <debug> mode
3) Hit [w][ret], followed by [s] for SAVE
4) Start address 1000; end address 4600; filename "CHAMP M/C"

Spectrum

1) LOAD"CHAMP"
2) Quit to BASIC : SAVE "CHAMP" LINE 1
3) SAVE "CHAMP M/C" CODE 27000,9231

The conditional branch instructions, as we have seen, depend on the contents of the processor status register. One reason for adding the binary display option to the Monitor program (see pages 118 and 198) was to enable you to inspect the contents of the PSR before and after an instruction is executed, and observe the changes in the flags. There is no single instruction in either 6502 or Z80 Assembly language to store the PSR contents, so we must use these commands:

Z80	
3E00 F5	PUSH AF
3E01 F5	PUSH AF
3E02 E1	POP HL
3E03 22 lo hi	LD (STORE1),HL
3E06 F1	POP AF
6502	
3E00 48	PHA
3E01 08	PHP
3E02 48	PHA
3E03 08	PHP
3E04 68	PLA
3E05 8D lo hi	STA STORE1
3E08 68	PLA
3E09 8D lo' hi'	STA (1+STORE1)
3E0C 28	PLP
3E0D 68	PLA

This sequence of instructions will cause the current contents of the PSR to be stored in the byte addressed by STORE1 (an address appropriate to your machine), while the accumulator contents will be stored at (1+STORE1). To use these instructions, simply insert them as a block before and after the program instruction whose effect you wish to observe. You must remember, however, to add two to the value of STORE1 every time you insert this block. When you've executed the program, you can use the Monitor to display the section of memory where you've stored the various contents of the PSR and the accumulator.

It may occur to you that this block should be treated as a subroutine rather than repeatedly entering it where it is required. There is an Assembly language equivalent of BASIC's GOSUB, but using it here would complicate matters since it uses the stack, and this would interfere with the block's use of the same list (PLA, PUSH, PHP, etc. are all stack manipulations, which will be more fully explained later). You may notice the difference in length between the Z80 and 6502 code: the Z80's two-byte registers and associated instructions are responsible for this variation.