BASIC string variable: When we write in BASIC:

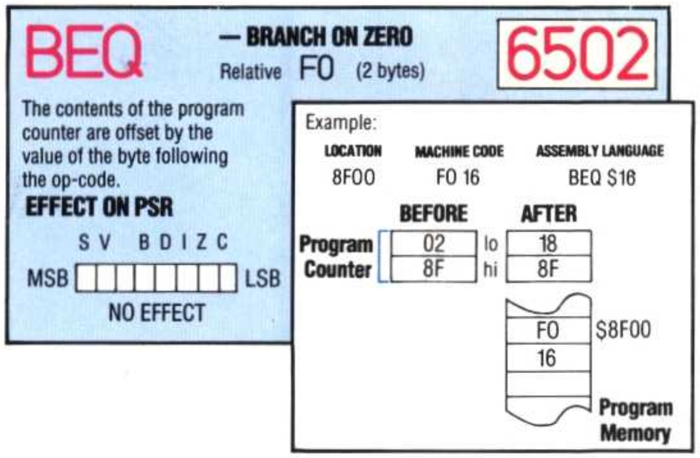
```
200 LET A$="MESSAGE 1"
```

then we are actually creating a pointer to the start of a table of bytes containing the ASCII codes for 'M', 'E', 'S', and so on. Whenever the BASIC interpreter encounters a reference to A$, it looks in its own symbol table to find the location at which it points — that is, the starting location of the contents of A$. Similarly, in our Assembly language program we can treat LABL1 as the equivalent of A$, given that we have already written a program fragment that allows us to manipulate a table using indexed addressing.

The pseudo-ops, then, allow us to remove absolute addresses and values from our programs, and replace them with symbols. This has the effect of diminishing the problems of portability and relocatability. What we need now is to be able to access these portable, relocatable modules from the main program. In other words, we need a machine code equivalent of BASIC's GOSUB command.

There is such an instruction, of course: JSR and CALL in 6502 and Z80 respectively. Both require an absolute address (which can be a label) as operand, and both have the effect of replacing the contents of the program counter with the address that forms their operand. The next instruction to be executed, therefore, will be the first instruction of the subroutine so addressed. Execution continues from that instruction until the RETURN instruction — RTS and RET respectively — is encountered. This command has the effect of replacing the current contents of the program counter with its contents immediately prior to the JSR or CALL instruction was executed. The next instruction to be executed, therefore, is the instruction immediately following the JSR or CALL. This is exactly the mechanism used by the BASIC interpreter in executing and returning from GOSUBs. It's easily understood as such, but it raises the question of how the old contents of the program counter are restored when the RETURN instruction is executed. The simple answer is that the JSR and CALL instructions first 'push' the program counter contents onto the stack (see illustration on page 136) before replacing them with the subroutine address; and the RTS and RET instruction 'pop' or 'pull' that address from the stack back into the program counter. The questions of what the stack is, how you push or pop it, and why you'd want to do so, are the subject of the next instalment of the course.

Instruction Set



BEQ — BRANCH ON ZERO 6502

Relative F0 (2 bytes)

The contents of the program counter are offset by the value of the byte following the op-code.

EFFECT ON PSR

S V B D I Z C

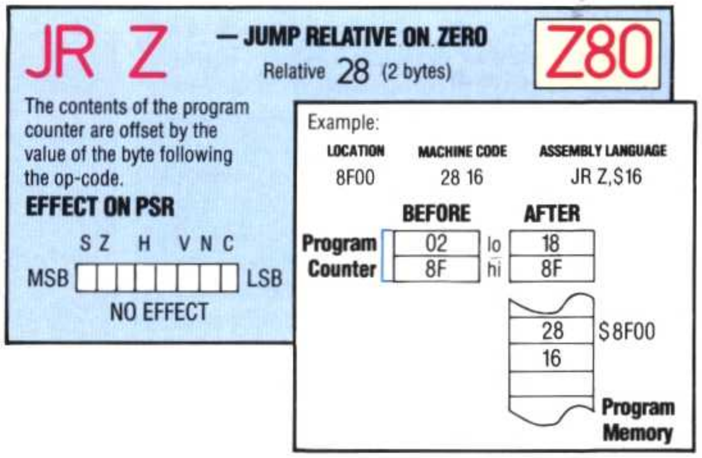
MSB [| | | | | | |] LSB

NO EFFECT

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
8F00	F0 16	BEQ $16

	BEFORE		AFTER
Program Counter	02	lo	18
	8F	hi	8F



JR Z — JUMP RELATIVE ON ZERO Z80

Relative 28 (2 bytes)

The contents of the program counter are offset by the value of the byte following the op-code.

EFFECT ON PSR

S Z H V N C

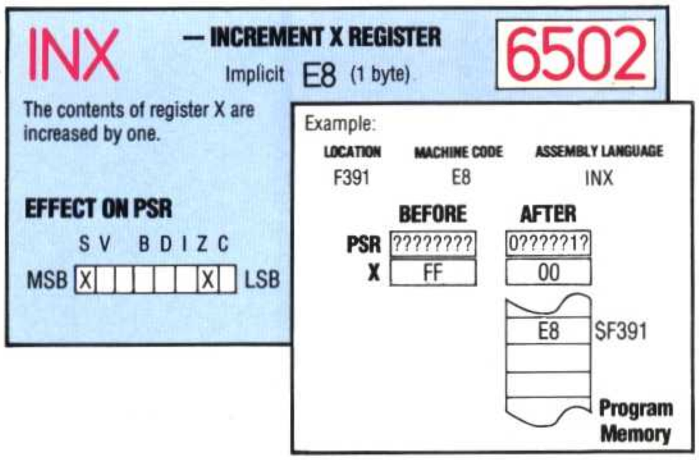
MSB [| | | | | | |] LSB

NO EFFECT

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
8F00	28 16	JR Z,$16

	BEFORE		AFTER
Program Counter	02	lo	18
	8F	hi	8F



INX — INCREMENT X REGISTER 6502

Implicit E8 (1 byte)

The contents of register X are increased by one.

EFFECT ON PSR

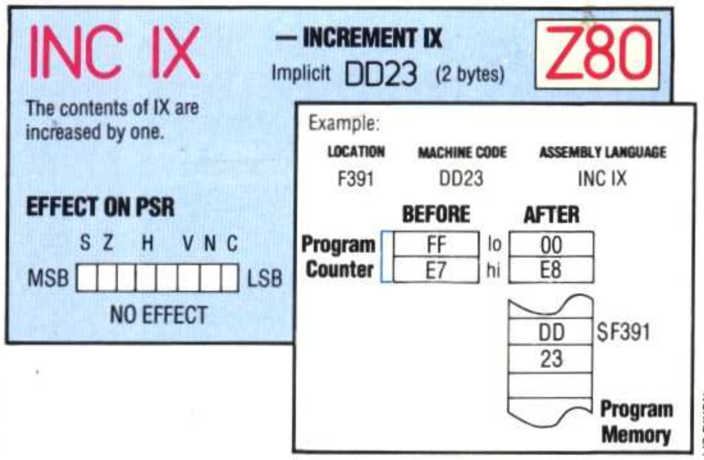
S V B D I Z C

MSB [X| | | | | |X|] LSB

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
F391	E8	INX

	BEFORE	AFTER
PSR	????????	0?????1?
X	FF	00



INC IX — INCREMENT IX Z80

Implicit DD23 (2 bytes)

The contents of IX are increased by one.

EFFECT ON PSR

S Z H V N C

MSB [| | | | | | |] LSB

NO EFFECT

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
F391	DD23	INC IX

	BEFORE		AFTER
Program Counter	FF	lo	00
	E7	hi	E8

LIZ DIXON