LAST IN FIRST OUT

The stack is a defined area of computer memory attached to the CPU that acts as a convenient workspace and takes a vital part in subroutine execution. It is easily accessed through the stack instructions, which permit the quick copying and restoring of register contents. We examine the stack and its operation in detail here.

Memory management is the essence of Assembly language programming, and most of the instructions we've studied so far in the course are concerned with simply loading data to or from memory locations. These locations have been accessed in a variety of ways — the addressing modes — but the instructions concerned have always taken a specific memory address as part of the operand. There is a class of instructions, however, that access a specific area of memory but do not take an address as operand. These instructions operate on the area of memory known as the *stack*, and they are known as the stack operations.

The stack is provided for both the central processing unit and the programmer to use as temporary workspace memory. It is a kind of 'scratch-pad', easily written to, read from and erased. The stack operations copy data from the CPU's registers into vacant areas of the stack, or copy data from the stack back into the CPU registers. These instructions require no address operand because a specified CPU register, the *stack pointer*, always contains the address of the next free stack location. Thus, anything written to the stack is automatically written to the byte pointed to by the stack pointer, and data loaded from the stack is always copied from the stack location last written to. Whenever a stack operation is executed, the stack pointer is adjusted as part of the operation.

In 6502 systems the stack is the 256 bytes of RAM from $0100 to $01FF; in Z80 systems the location and size of the stack are determined by the operating system, but may be changed by the programmer. This variation reflects the differences in the internal organisation of the two microprocessors (see the diagram on page 136): the 6502 has a single-byte stack pointer, while the Z80 stack pointer consists of two bytes.

The contents of the 6502 stack pointer are treated by the CPU as the lo-byte of the stack address, and a hi-byte of $01 is automatically added to this by means of a 'ninth bit' wired into the stack pointer. This extra bit is always set to one, so 6502 stack addresses are all on page one.

The Z80 stack pointer is a two-byte register capable of addressing any location between $0000 and $FFFF — the entire address space of the Z80 itself. The stack can thus be located anywhere in RAM, and its location can be changed by the programmer. This is not recommended, however, since the operating system initially sets the stack location and stores data on it. As the operating system may interrupt the execution of any machine code program at any time, and expect to find data relevant to its operation on the stack, any alteration of the location of the stack will mean that the data will not be available to it and the system may crash.

As an example of the use of the stack, consider the following routine to exchange the contents of two memory locations, LOC1 and LOC2:

6502		Z80	
LDA	LOC1	LD	A,(LOC1)
PHA		PUSH	AF
LDA	LOC2	LD	A,(LOC2)
STA	LOC1	LD	(LOC1),A
PLA		POP	AF
STA	LOC2	LD	(LOC2), A

The contents of LOC1 are loaded into the accumulator, and from there copied or 'pushed' onto the stack. The contents of LOC2 are then loaded to the accumulator, and stored in LOC1. The contents of the top byte of the stack are then copied or 'popped' into the accumulator, which restores the original contents of LOC1 to the accumulator. This is copied to LOC2, and the exchange is complete. Notice that the stack operations 'saved' the contents of LOC1 in memory as long as needed, but without the program specifying any memory location — except, by implication, the next free location on the stack.

This program fragment shows us a lot about stack operations. Primarily, they are reciprocal and sequential. The last item pushed onto the stack is retrieved by the next pop from the stack. Successive pushes with no intervening pops write data into successive stack locations, one 'above' the other, while pops without intervening pushes access successive locations 'downwards' from the current 'top' of the stack.

To visualise the stack, imagine writing notes on postcards and stacking them next to you on the desk, then reading and discarding cards until the stack is empty. The most recently written of the cards remaining in the stack is always the one on top. For this reason the stack is known as a Last In First Out (LIFO) data structure. Its converse, a First In First Out (FIFO) structure, is a queue. It