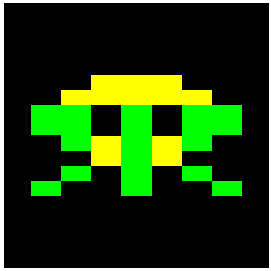


6

THE STACK

The 6502 processor supports a hardware stack. This is an important part of the computer which can be used both by the CPU and the programmer. This chapter looks at how the stack can be used and how it performs its task.

6.1 Using the stack

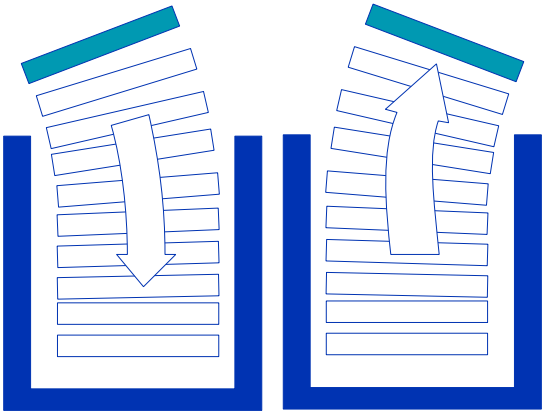
A hardware stack is simply a set of memory locations (&100 to &1FF) which are reserved by the processor. These locations can be used as temporary storage locations. Up until now, when we have wanted to store a value, we might have used.

```
STA tempaddr
```

And to recall the value which was in the accumulator at that time, the instruction

```
LDA tempaddr
```

Loading the accumulator with the value from 'tempaddr' does not alter the value stored in 'tempaddr'. Hence the number may be recalled several times. When storing a value on the stack, however, the situation is different. The value to be stored is 'pushed' onto the stack and when it is



last-in first-out

'Last-in, first-out'; the stack forbids the use of any item other than in that order.

wanted again, it is ' pulled'off into a register - a one-time only operation.

The stack is a LIFO structure, these initials stand for ' last-in,first-out' ,which means that the first item put on the stack will be at the bottom and so will be the least readily available, whereas the last item on the stack will be at the top and will be the first one to be pulled off it.

The four instructions which a programmer can use to access the stack are:

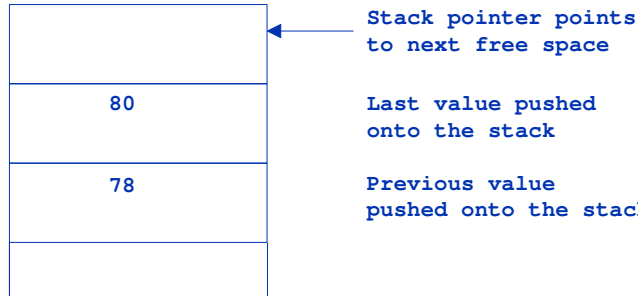
Mnemonic	**Description**
PHA	push the contents of A onto the stack
PLA	pull a value off the stack and store it in A
PHP	push the contents of the status register P
PLP	pull a value off the stack and store it in P

A stack pointer is used to manage the stack. This is a register which contains the address of the top location being used. For example, on encountering a PLA instruction, the accumulator is loaded from the memory location pointed to by the stack pointer and then the stack pointer is automatically moved back one location. Whenever a PHA instruction is

executed, the accumulator is stored in the memory location pointed to by the stack pointer and then the stack pointer is moved on to the next location.

Example
The following series of pushes and pulls leaves the stack in a state which is shown below:

```
LDA #78
PHA
LDA #79
PHA
PLA
LDA #80
PHA
```



Note that the value 79 which was pushed onto the stack and then pulled off it again no longer occupies a location on the stack.

To see why the stack is used as a temporary storage place in preference to a memory location such as tempaddr' ¸consider the two alternative sections of assembler below:

```
PHA                 STA tempaddr
LDA addr            LDA addr
```

```
CLC                 CLC
ADC #12             ADC #12
STA addr            STA addr
PLA                 LDA tempaddr
```

These both produce the same result when executed but the one on the left will produce less code. This is because the LDA and STA instructions consist of either two or three bytes, one for the op-code and one or two for the address. The PHA and PLA instructions, however, just consist of an op-code.

Note that if more than one value is stored on the stack at once, care must be taken when these values are retrieved. Because it is a LIFO structure, the values must be taken off the stack in the opposite order to how they were placed on it, e.g.

```
PHA     save accumulator
TXA     prepare to save X
PHA     save X
.
.
PLA     restore value
TAX   transfer to X the last vaLue pushed
PLA     restore accumulator
```

When using the stack it is very important to pull as many values as you push. Otherwise confusion can arise as we will see below.

6.2 How the CPU stores addresses

The stack is used by the CPU as well as by the programmer. On encountering a JSR instruction, the address of the instruction following the JSR is stored so that the CPU knows where to start executing from when it comes to the end of the subroutine. This is done by pushing the two bytes of the address onto the stack so that they can be retrieved when the RTS is reached. Thus a routine which is entered with a JSR and finishes with RTS should always pull the same number of bytes as are pushed otherwise the value obtained from the top of the stack by the RTS will not be the correct return address, e.g.

```
.entersubroutine
```

```
PHA
LDA addr
CLC
ADC #12
RTS
```

When the RTS is reached the CPU will pull two values off the stack, and put them into the program counter. However, as one of these values is the value pushed with the ' PHA' the program counter will almost certainly contain the wrong address. This will mean that the CPU will start trying to execute instructions at the wrong address and do something undefined by the designers of the 6502.

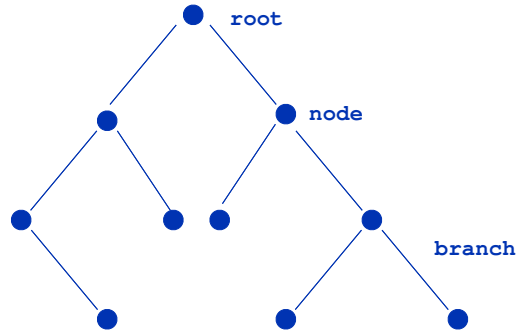
6.3 Recursion

One of the most important reasons for using a stack to hold the addresses of subroutine returns is that recursion is then automatically supported.

A recursive subroutine is one which calls itself. This can be a very powerful feature and enables a programmer to implement tree structures as shown below.

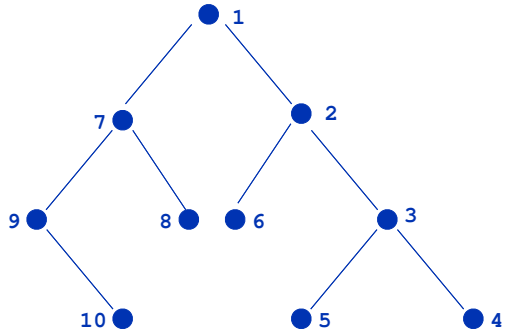
Using trees

A tree in computing is normally pictured as follows:



Note that it is usually drawn with the ' root' at the top.

In order to print out all the elements (root and nodes) in the tree, you must write a routine which prints out an element and then goes down a branch to the element beneath it. If there isn' tan element below, then it goes back up one level and sees if there is an alternative branch from there. For the above tree the order that the elements would be visited, assuming that the routine shows a preference for right-hand branches, is:



A BASIC routine to do this is as follows:

```
DEFPROCtree (element)
PRINT value(element)
IF right(element) THEN PROCtree(right(element))
IF left(element) THEN PROCtree(left(element))
ENDPROC
```

This assumes that each element has three things known about it: its value, the element that its right branch leads to (FALSE if no branch) and the element that its left branch leads to (FALSE if no branch). These should be stored in three arrays whose names are ' value' , ' right' and ' left' .

The routine can be called using

```
PROCtree(0)
```

The assembler version of this is:

```
.enter
  LDX #0              Start at root (zeroth element)
.tree
  LDA value,X         Get value of element
  JSR printnumber     See section 13.1 for details
  LDA right,X         Is there a right branch?
  BEQ tryleft         If not, try a left one
  TAX
  JSR tree            Else take that branch
.tryleft
  LDA Left,X          Is there a left branch?
  BEQ backup          If not, go back up one level
  TAX
  JSR tree backup     Else take that branch
  RTS                 Return
```

In this case the block of memory locations starting with the address 'value' should contain the values of each element in turn: those starting at 'right' should contain the element to which each one's right-hand branch leads, and 'left' the element to which each one's left-hand branch leads.