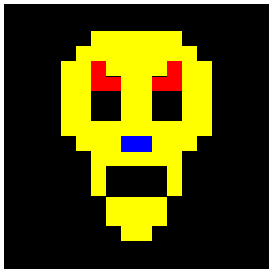


4

LOGICAL OPERATIONS, SHIFTS AND ROTATES

We have seen previously that each byte or memory location is made up of eight bits, each of which can be set to the value 0 or 1. Although the operations we have considered so far have treated the whole byte as the smallest quantity being dealt with, many operations in the computer's instruction set are best considered as operations which act on eight separate bits. Some of these perform such important tasks as changing the case of characters, or multiplying and dividing.

4.1 Logical operations

Logical operations are performed between the individual bits of two operands; one of the operands is always the accumulator and the other is a memory location or immediate value. In this section three such operations are introduced; AND, OR and EOR. A truth table is used to give a compact description of each operation. This takes two single bit inputs which, for convenience, we call A and B, and shows the bit which is produced as a result of ANDing, ORing or EORing them together. This is known as Boolean logic after its inventor, George Boole.

AND

Mnemonic	**Description Symbol**
AND	**AND** accumulator with
memory	A=A AND M

Truth table:

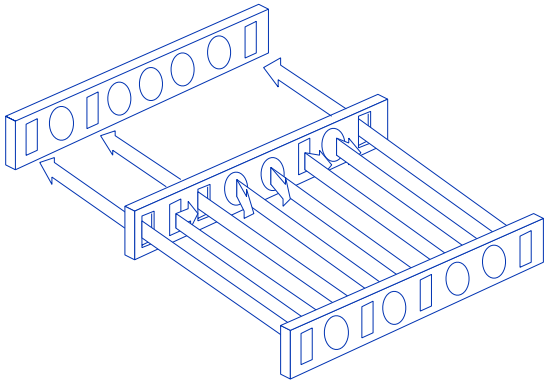
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

The AND operation sets a bit of the result to a 1 only if the corresponding bit of one operand is a 1 AND the corresponding bit of the other operand is a 1; otherwise the bit in the result is a zero, e.g.

```
                  Hexadecimal          Binary

operand 1            A9           1 0 1 0 1 0 0 1
operand 2            E5           1 1 1 0 0 1 0 1
                     --           ---------------
result of AND        A1           1 0 1 0 0 0 0 1
                     --           ---------------
```

One way of thinking of the AND operation is that one operand acts as a 'mask', and only where there are ones in the mask do the corresponding bits in the other operand 'show through'; otherwise, the bits are zero.



OR

Mnemonic	Description	Symbol
ORA	**OR** accumulator with memory	A=A OR M

Truth table:

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

The OR operation sets a bit of the result to a I if the corresponding bit of one operand is a 1 OR the corresponding bit of the other operand is a 1, or indeed, if they are both ones; otherwise the bit in the result is zero, e.g.

	Hexadecimal	Binary
operand 1	A9	1 0 1 0 1 0 0 1
operand 2	E5	1 1 1 0 0 1 0 1
	--	---
result of OR	ED	1 1 1 0 1 1 0 1
	--	---

Exclusive-OR

Mnemonic	Description	Symbol
EOR	Exclusive-**OR** accumulator with memory	A=A EOR M

Truth table:

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

The Exclusive-OR operation is like the OR operation, except that a bit in the result is set to I only if the corresponding bit of one operand is a 1, or if the corresponding bit of the other operand is a I, but not if they are both ones, e.g.

```
          Hexadecimal              Binary

operand 1          A9          1 0 1 0 1 0 0 1
operand 2          E5          1 1 1 0 0 1 0 1
                   --          ---------------
result of EOR      4C          0 1 0 0 1 1 0 1
                   --          ---------------
```

Another way of thinking of the Exclusive-OR operation is that a bit of the result is 1 if and only if the corresponding bits in the operands are different.

Example - converting lower to upper case
The following example converts all characters entered in lower case to upper case. See Appendix A for the ASCII character set.

```
.loop
   JSR osrdch          Get character
   AND #&DF            Make case bit zero
   JSR oswrch          Print it
   JMP loop            And do it again
```

Try altering this using the ' ORA'instruction to convert all characters to lower case. When you have succeeded in doing this try writing a routine to swap case.

4.2 The BIT instruction
This instruction is available to test whether individual bits of a number are set or not.

Mnemonic	**Description**
BIT	Compare memory bits with accumulator

The instruction AND' sthe bits of the accumulator and the memory. The zero and negative flags are set

or cleared as a result of this operation; Z=1 if the result was 0 and N = top bit, V = bit 6 of contents of location.

Hence BIT may be used to test any bit of the memory by loading the accumulator with a value containing a 1 in the relevant position and 0' severywhere else. Then the values 0 and 1 for Z show whether the bit was or was not set respectively, e.g.

```
LDA #4          4=00000100
BIT addr
BEQ bit-not-set
```

If addr contained, for example, &43 (01000011) then the branch would occur. If, however, addr contained &44 (01000100) then the branch would not take place. Bit differs from the AND instruction in that it does not corrupt the accumulator.

4.3 Rotates and shifts

The rotate and shift operations move the bits in a byte either left or right.

Mnemonic	Description
ASL	arithmetic shift left
ROL	rotate left
LSR	logical shift right
ROR	rotate right

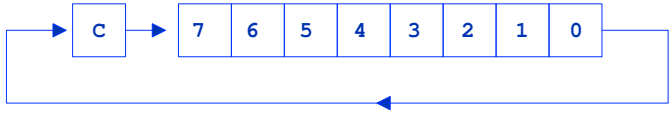
The ASL instruction moves all the bits one place to the left; what was the high-order bit is put into the carry flag, and a zero is put into the low-order bit of the byte. The ROL instruction is identical except that the previous value of the carry flag is put into the low-order bit instead of zero.

The right shift and rotate right instructions work in a similar way except that the bits are shifted to the right.

ASL - Arithmetic shift left one bit

LSR – Logical shift right one bit

ROR – Rotate right one bit



Example -- multiplying by two

The most efficient way to multiply a two-byte number, stored in ' addr' and ' addr + 1' , by two, is to shift the contents of the two bytes one place to the left. Where ' addr' and ' addr+1' are the addresses of the locations storing the low and high bytes of the number being doubled, use the following two statements:

```
ASL addr
ROL addr + 1
```

This works by using the carry to hold the bit that falls off the end of ' addr' , and then using the ROL statement, which puts the carry into the correct place in the high-order byte.