MEAN machine CODE

What does Machine Language mean to you? Is it that strange mid-Atlantic drone that Speak and Spell games used? Or the key to the true power of your C64? Jason Finch reveals all...

Part 4

Hard as it may be to believe, by the end of the next page you will have learnt the facts about another 17 machine language commands. It's true. Because this month I will guide you gently through the realms of arithmetic, logical, transfer, shift and rotate instructions. Not only that, but I'll tell you all about the incredibly easy zero page and relative addressing modes. The only question left to ask is where to begin? Let's get that answered right away.

ADDRESSING IT

Remember that last month I told you what the zero page was? If you don't – because you didn't read the mag (in which case don't tell Dave because he'll only use it an excuse to plug Back Issues) or you've just got a plain bad memory – don't worry. It's just a posh name for the first 256 memory locations. Zero page addressing mode allows you to tell the computer what address you want by supplying only one byte of info, not the normal two. Like `LDA, $D020` requires two bytes, stored as `$20` and `$D0` in the memory for reasons that elude even the illustrious Dave. But in zero page, the high byte of every address is `$00` and so we just get rid of it. So if you stored a value in location `$00FB`, you access it by doing `LDA $FB` instead of `LDA $00FB`. You can do that with practically everything where you normally use two bytes. Take, as a simple example, the following short assembly language program:

```
100 *=49152
110 LDA 1064; $0428
120 STA 251; $00FB
130 RTS
```

Line 110 uses absolute addressing and is stored in memory as `$AD, $28, $04` which is three bytes. Line 120 uses zero page addressing because the memory location in question (251) lies in the first 256. It is stored as `$85, $FB` which is only two.

Relative addressing is not used in many instructions; in fact you will only come across it in branch instructions. It is the equivalent of saying, "take two steps backwards from here," or, "take three steps forwards from here". An actual address is not stored in memory, just the number of bytes either forward or back that the program must jump past. For convenience you always specify the address when you enter assembly language like `BNE LOOP` or `BNE $C000`, but these are never stored literally.

CARRY THE CAN

In the auspicious world of the C64, flags are things which are either on or off. Like light bulbs. And in the said world there is no concept of a dimmer switch so don't get funny with me, matey. If you want a flag to be on, you set it; otherwise you clear it.

For now I'll just tell you about the carry flag. It's used a lot in machine code; during additions, subtractions and even during branch instructions. In fact, the instruction to add two numbers together is `ADC` which stands for ADd with Carry. Similarly, `SBC` means SuBtract with Carry. To clear and set the flag yourself, you do `CLC` or `SEC`. `CLC`, quite unsurprisingly, is short for CLear Carry and unless you're a big, fat lemon I don't need to say what `SEC` stands for. The relevant branching instructions are `BCC` and `BCS`: Branch if Carry Clear and Branch if Carry Set.

```
100 *=49152
110 SEC
120 BCC CLEAR
130 RTS
140 CLEAR ;
150 INC 53280 ;BORDER COLOUR
160 RTS
```

Try assembling the above program and executing it with `SYS 49152`. You should find that nothing happens when you run it. Great program, eh? But hang on. Change the `SEC` in line 110 to `CLC` and run it again. Now you've cleared the carry flag and so the branch will take place and the border colour should change. Experiment using `SEC` and `CLC` with either `BCC` or `BCS`.

IT ALL ADDS UP

When you add two numbers together in machine language, the computer very thoughtfully adds one to the result if the carry flag is set. Therefore it is advisable to clear it beforehand. On the other hand, you should set the carry flag before a subtraction. Because the highest number you can store in one byte is 255, the computer will automatically set the carry flag during an addition if the result is bigger than 255. That's why it's called a carry flag; the high byte is carried over. If you add 3 to 5 the result is 8, which is fine in decimal. If you add 5 to 8 you get 13. The first time you ever did this you will have been told that you have to 'carry' the one over into the tens column and write a three in the units column. It's the same thing.

In theory it is very simple to write a program which will read in a number from a memory location, change it and then store the new result back. Try out the following program which puts the theory into practice. As you should be doing with all these listings, experiment with the commands and values to see what happens:

```
100 *=49152
110 LDA 251
120 CLC
130 ADC #25
140 STA 251
150 RTS
```

Assemble the program and enter `POKE 251,100:SYS 49152:PRINT PEEK(251)`. You should get the result of 125 because line 130 adds 25 on to the value read in by

COMMAND SUMMERY:

This month's Mean Machine Code was bought to you by the letters M and C and the commands...

ADC	Add value to accumulator
SEC	Subtract value from accumulator
AND	Logically AND value with accumulator
ORA	Logically OR value with accumulator
EOR	Logically EOR value with accumulator
TAX	Transfer accumulator to X register
TXA	Transfer X register to accumulator
TAY	Transfer accumulator to Y register
TYA	Transfer Y register to accumulator
ASL	Shift bits left one bit
ROL	Roll bits left one bit
LSR	Shift bits right one bit
ROR	Roll bits right one bit
CLC	Clear the carry flag
SEC	Set the carry flag
BCC	Branch if carry flag is clear
BCS	Branch if carry flag is set

line 110. Line 140 stores it back. Change the CLC to]SEC and the ADC to SBC. Observe the fascinating fact that the result is now 75. Heavens.

FROM HERE TO THERE

Transferring is always very handy (*unless it involves money in which case it always seems to be unnecessarily complicated – Dave speaking from bitter experience*). It normally means moving something from one place to another different place. In our case, it means copying a value from one register into another one; not actually moving it. So, to read in the value in the accumulator and to then write it to the X register, you could do something like STA $FB followed by LDX $FB which would store the value in the accumulator to memory and then read it back into the X register. Alternatively, you could use TAX. Do I really have to tell you what it means (and no stupid comments about the Inland Revenue from you, Dave)? Transfer the Accumulator to the X register. There is also TAY which shoves it in the Y register, and TXA and TYA which do the reverse; TXA takes whatever is in the X register and copies it to the accumulator.

ROMARKABLE

I shall take this apt opportunity to thrust more ROM routines into your face.

- JSR $E544 will clear the screen
- JSR $E566 will home the cursor
- JSR $E097 will cause a random number to be generated which can then be read by doing a quick LDA $8F.

The best two sets of ROM routines are those demonstrated by the routine below.

```
100 *=49152
110 JSR $E20E
120 JSR $AD9E
130 JSR $B7F7
140 ;
150 LDA 21
160 LDX 20
170 JSR $BDCD
180 ;
190 RTS
```

The three ROM routines in lines 110 to 130 are very useful indeed if you are mixing Basic and machine language. They read in a value after a comma and store the result in locations 20 and 21 of zero page. The low byte is in 20 and the high byte in 21 for everything is stored in lo-hi byte format. The nifty ROM routine at line 170 displays a decimal number to the screen which must be supplied in the rather inconvenient lo-hi byte format. The high byte must be given by the accumulator and the low byte by the X register. That is why the values are transferred across in lines 150 to 160. I trust you will find a use for these. For example, amend the simple addition proglet as shown below. When you have assembled it enter SYS 49152,100 to marvel at what happens.

```
100 *=49152
110 JSR $E20E
120 JSR $AD9E
130 JSR $B7F7
140 LDA 20
150 CLC
160 ADC #25
170 TAX
180 LDA #0
190 JSR $BDCD
200 RTS
```

IT'S LOGICAL

That was the paragraph header I used way back in *CF40* if my memory is working. That's the issue that described the logical AND and OR operators. It followed *CF38*'s introduction to binary and *CF39*'s information on how memory works. You should have read and understood all three of those if you want any hope of understanding the rest of this series (*yes, yes, yes... at last I get the chance to plug the Back Issues service on page 17 – Dave*). The equivalent of the Basic AND command is, quite spookily, AND. The equivalent of OR is, drum roll please, ORA. Your straightforward conversion of A=PEEK(56320):IF (A AND 16)=16 THEN.. would be:

```
100 *=49152
110 LDA 56320
120 AND #16
130 CMP #16
140 BEQ ...
```

I didn't cover EOR in *CF40* because Basic doesn't have such a thing. It's referred to as Exclusive-OR. In simple terms, it flips bits between zero and one. The good old-fashioned truth table looks something like:

```
0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0
```

You may find it easier to look at a couple of examples with the decimal and binary equivalents, just like I showed for AND and OR in *CF40*. The decimal value is usually irrelevant to the proceedings. When you grasp a bit more about bits and stuff, you will begin to understand why.

Binary		Decimal
00100110		38
00001111	EOR	15
--------		--
00101001		41
00101001		41
01010101	EOR	85
--------		--
01111100		124

ROLLING AND SHIFTING

Believe it or not, rolling and shifting are both dead complicated if you want them to be, or both incredibly easy if you prefer it that way (and who doesn't?). It all depends on how you look at them. If you're upside down with one eye closed, they're particularly bad.

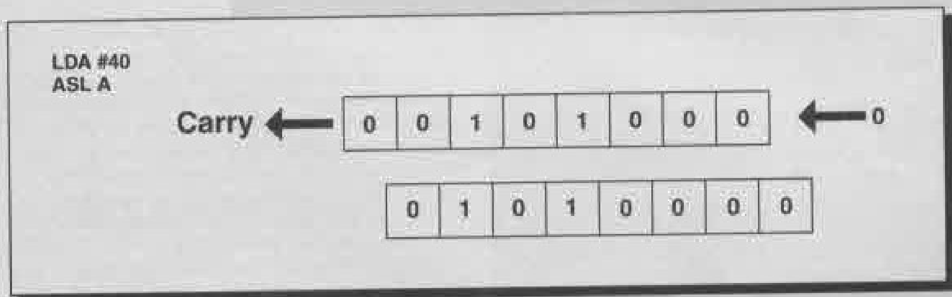
Remember that a byte consists of eight bits. If you imagine shoving them all one bit to the left and filling the empty bit with a zero, you have imagined shifting. The bit that disappears out of the left hand end becomes the carry flag; clear if the bit was a zero, set if it was a one. Rolling involves a similar thing, but the contents of the carry flag are bunged in the empty bit before the carry flag is filled with the whatever was in the left-hand end. The whole thing is much easier to show you in a diagram than in a picture. Check out the diagrams lovingly prepared by art editor extraordinaire, Ollie (*what are you after? – Ollie*) which show shifting and rolling to see what I mean. Doing the same things but going right instead of left is just known as shifting right instead of shifting left. Nothing is too hard in this world.

The instructions that are relevant here are ASL (Arithmetic Shift Left), LSR (Logical Shift Right), ROL (Roll Left) and ROR (Roll Right). Shifting the bits to the left, assuming none are lost off the left-hand end, is equivalent to multiplying the number by two, and shifting them right is the equivalent of dividing by two. You don't believe me? I'm hurt. But if you want proof try this one out:

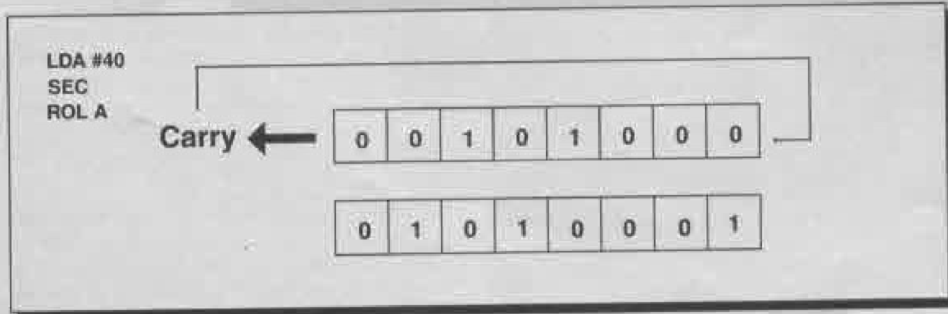
```
100 *=49152
110 LDA #40
120 ASL A
130 STA 251
140 RTS
```

After assembling and running the code with SYS 49152, you should get the answer 80 if you PRINT PEEK(251). Change the ASL to LSR and you'll get 20. I should point out that you are using the accumulator addressing mode here. The 'A' after the instruction means, "Do this on the accumulator". You could equally do it to a memory location such as ASL $0400 or LSR 53280. Whatever takes your fancy.

- By the way, Dave asked me to remind you (probably because he's trying to flog off a few back issues) that we gave away an Assembler on *CF46*'s Power Pack. Got that? No? Then get it.



Above is shifting, below is rolling. Sound like some rubbish TS Eliot would have written, doesn't it?



NEXT MONTH

Put on your hard hats, because indexed addressing is coming up next month. I'm also going to show you how you can use the instructions I introduced this month in order to move a sprite around the screen with a joystick. This weird thing called the stack will also appear. Until then, keep playing with your assembler, and remember – know your code and code your nose.