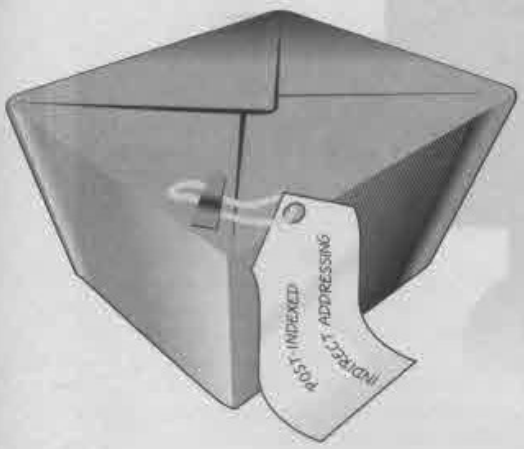
MEAN machine CODE

We're into the heavy stuff this month with our Machine Code series, so if you're of a nervous disposition or pregnant, Jason Finch advises that you take extreme caution...

This month we're going to take a brief trip to hell and back. That's right, it's time for post-indexed indirect addressing (*sounds like the scientific name for, "the cheque's in the post" – Dave*). But don't faint just yet because I'm also going to introduce you to the joys of pre-indexed indirect addressing. If you're still in the land of the living after that, we'll take a brief look at the stack. But before all that I'll divulge a couple of secrets about the BCC and BCS branch instructions that were covered last month.



MORE OR LESS

There may come a time when you want to compare one number with another to check which one is the biggest. You'll probably know about less-than and greater-than signs in Basic. They mean you can do things like:

```
10 A=PEEK(1024)
20 IF A<65 THEN GOSUB 50
```

Remember that whenever you have to say "IF this THEN do that" you need one of the eight branch instructions in machine code. We've only covered four so far, but those four (BEQ, BNE, BCC and BCS) are the most common. To do a less-than comparison like in the snippet of program above, you have to use BCC:

```
10 LDA 1024
20 CMP #65
30 BCC SUBROUTINE
```

You don't really need to know why this is the case, but I'll tell you anyway. When the comparison is done, the computer subtracts from the value you are comparing, the value you are comparing it with. You know that if this results in a negative answer being produced, the carry flag will be cleared. Much the same as if you were to do the following:

```
10 LDA 1024
20 SBC
30 SBC #65
40 BCC SUBROUTINE
```

To check if one value is greater than another you use BCS instead. It actually checks if it is greater than or equal to the number concerned. Therefore the equivalent of `A=PEEK(1024):IF A>=65 THEN...` in Machine Code would be:

```
10 LDA 1024
20 CMP #65
30 BCS SUBROUTINE
```

You may be pondering over how you check if a value is only greater than a number. It's simple, you just eliminate the possibility of it being equal beforehand:

```
10 LDA 1024
20 CMP #65
30 BEQ CARRYON
40 BCS SUBROUTINE
50 CARRYON ;
```

MULTIPLE MATHS

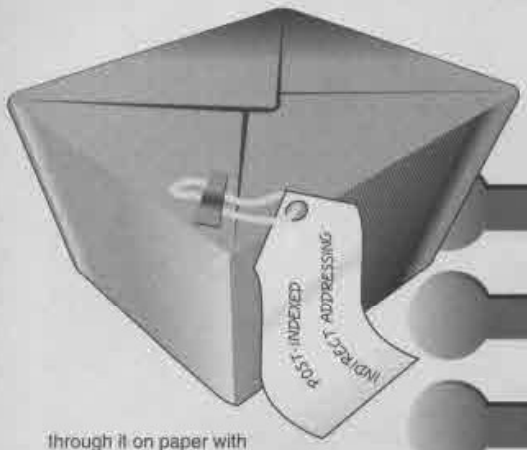
There are instructions for adding and subtracting in machine code. You saw those last month. You also saw rolling and shifting; the equivalent of multiplying and dividing a number by two. But what if you want to do something a bit more challenging? Say you wanted to multiply 485 by 57 in machine code. The answer is to use a multiplication routine that someone has already made for you:

```
1000 MULTIPLY ;
1010 LDA #0
1020 STA RESULT
1030 STA RESULT+1
1040 LDY #8
1050 MULT1 ;
1060 LSR MULTIPLIER
1070 BCC MULT2
1080 LDA RESULT
1090 CLC
1100 ADC MULTIPLICAND
1110 STA RESULT
1120 LDA RESULT+1
1130 ADC MULTIPLICAND+1
1140 STA RESULT+1
1150 MULT2 ;
1160 ASL MULTIPLICAND
1170 ROL MULTIPLICAND+1
1180 DEY
1190 BNE MULT1
1200 RTS
```

This may all seem quite complicated at first... er, that's because it is! You have to use lo-hi byte notation with MULTIPLICAND. The MULTIPLIER is a single byte number (0-255) with the lo-hi byte result being given in the memory locations that you have assigned to RESULT. To understand why it works you will need to understand how bits are manipulated. Perhaps you should work

STICKY SITUATION

As part of the *Techie Tips Proggy Selector* this month, you will find a Basic loader and the equivalent assembly language source code for a routine that moves a sprite around the screen controlled by a joystick in port two. You call it by doing SYS 49152,X,Y where X and Y are the horizontal and vertical speeds (in pixels) respectively. I want you to read through the source code and work out why everything occurs as it does. There'll be a test on it next month.



through it on paper with some simple examples like seven times three to get the hang of the way it works.

INDEXED ADDRESSING

It will appear to be your worst nightmare. It's worse than a soggy sponge cake, and definitely worse than maths homework. Yes, it's post-indexed indirect addressing. The trouble is that it's dead useful and you will need it a lot, so it would help if you do at least try to understand it. Before I try to explain it, let's look at an example:

```
100 *=49152
110 LDA #$00; = LDA #<$D000
120 STA 251
130 LDA #$D0; = LDA #>$D000
140 STA 252
150 LDY #0
160 LOOP ;
170 LDA (251),Y
180 STA 1024,Y
190 INY
200 CPY #16
210 BNE LOOP
220 RTS
```

The post-indexed bit is line 170. The LDA instruction (without a hash sign) reads the contents of a memory location into the accumulator. Normally you would do LDA 1024 or similar. This tells the computer right away which address you are interested in. The 251 in brackets on line 170 is also an address, one in zero page. The computer looks at what's in 251 and 252, the next location, and makes up a new number from this. In our example, we have stored $00 in 251 and $D0 in 252 (in lines 110-140). Therefore, the new number would be $D000. Locations 251 and 252 therefore act as a vector, storing the "base" address in standard low byte followed by high byte format. The contents of the Y index register are then added to this base address to find the actual address. You must always use a zero page address as vector, and always the Y index register. Try this one:

```
100 *=49152
110 LDA #$E8; = LDA #<$03E8
120 STA 251
130 LDA #$03; = LDA #>$03E8
140 STA 252
150 LDY #$18
160 LDA (251),Y
170 INY
180 STA (251),Y
190 RTS
```

STACKING

No, I'm not talking about your Saturday job at the local supermarket. The C64's stack is the first page of memory after zero page; locations $0100 to $01FF inclusive. It works pretty much like a stack of books. You add things to the stack and can then take them off again later; the last thing added to the stack is the first thing that is removed. The computer uses the stack to store memory locations during JSR instructions and the like. This is how it remembers where to go when it reaches an RTS at the end of a routine. You can use it as well. There are four commands related to adding items to the stack (pushing) and removing them (pulling). You can push or pull the accumulator or the processor status register using the PHA, PLA, PHP and PLP instructions. We'll leave PHP and PLP until I've explained what the processor status register is.

```
10 *=49152
20 LDA 1025
30 PHA
40 LDA 1024
50 STA 1025
60 PLA
70 STA 1024
```

The above short piece of code swaps the contents of locations 1024 and 1025 by using the stack. It reads in the value in 1025 into the accumulator. This value is then pushed on to the stack. Imagine it as writing the number on the front of a book and then putting that book on to your pile of books. The value in 1024 is then read in and stored back to 1025. The accumulator is then pulled back off the stack and stored at location 1024. This is the equivalent of retrieving the book on the top of the pile and looking at the number written on the front of it. In any one routine or subroutine, the number of PHAs must match exactly the number of PLAs. If they differ, the computer will get confused and crash. For example, never do something like this:

```
10 LDA 53280
20 PHA
30 JSR ROUTINE
40 RTS
50 ROUTINE ;
60 PLA
70 STA 53281
80 RTS
```

Line 60 will not pull the same value off the stack as was pushed on to it at line 20. Line 30 results in a two byte address being pushed on to the stack so that the computer knows where to go back to (the return address) at line 80. Therefore you actually destroy the return address by pulling part of it off the stack in line 60.

Lines 110-140 store the two-byte number $03E8 as a vector in locations 251 and 252. Line 160 tells the computer to look at locations 251 and 252 for a two-byte number, and it finds $03E8 because that's what you've just stored there. It adds the Y index register to this: $03E8+$18=$0400 (1000+24=1024 in decimal). So 1024 is the address from which it loads a value into the accumulator. The Y register is then incremented at line 170 and line 180 stores the result back to an indexed address. This time it is $03E8+$19=$0401.

FILL 'ER UP

You can use post-indexed addressing to fill large chunks of memory with the same information very quickly. For example, the following piece of code will clear the standard bitmap ($2000-$3FFF) area in one foul swoop:

```
100 *=49152
110 LDA #$00;
120 STA 251 ; STORE $2000 AS A
130 LDA #$20; VECTOR AT 251/252
140 STA 252 ;
150 LDY #0
160 TYA ; COULD DO LDA #0
170 LOOP ;
180 STA (251),Y
190 INY
200 BNE LOOP; LOOP FOR LO-BYTE
210 INC 252 ; INCREMENT HI-BYTE
220 LDA 252 ; CHECK HI-BYTE
230 CMP #$40; UP TO $4000
240 BNE LOOP
250 RTS
```

BEFORE IT ALL

You can do pre-indexed indirect addressing as well. It is very rarely used in the normal course of things, but I will cover it to make sure your fountain of knowledge has enough water in it. With this, you still use a zero page address as a vector, but it works with the X index register. We are also still dealing with reading values from memory addresses rather than loading them directly as numbers using a hash sign.

```
10 LDA #$00
20 STA 253
30 LDA #$D0
40 STA 254
50 LDX #2
60 LDA (251,X)
```

What happens here is that the computer thinks, "Ah, location 251. But hang about, I need to add the X register to that first. Only then can I use the right vector." So that's what happens. Here, X=2 and so 251+X=253. So now we use that as a vector and must look at locations 253 and 254. Here we have stored, in the standard format, the two byte number $D000. Therefore, our LDA (251,X) in the above example actually does the same as LDA $D000. There are a few places where this sort of thing will prove necessary. When you need to use it, you will suddenly realise after a bit that it's the only way; you won't find any uses by thinking about it (*is that why you can't come up with any examples? – Dave*).

NEXT MUNTH

Let the nation mourn, for next month is the final episode of Mean Machine Code. We'll be taking a detailed look at the sprite program from this month's Power Pack, and I'll clear up everything that hasn't yet been explained. Don't have nightmares about that indexed addressing!