Machine Code Programming

Programming Machine Code on the C64

See also: programming sprites on the C64

One of the criticisms leveled against the old Commodore 64 is that its built-in BASIC programming language was rather minimalistic, that is it had a shortage of commands and programming many tasks required POKE and PEEK or even machine code. Whilst such criticisms have a certain validity, this 'weakness' was in many ways the strength of the C64. Programming the 8-bit 6510 processor of the C64 encouraged the programmer to obtain an intimate knowledge of the computer's memory and also of its operating system kernal (i.e. kernel, once misspelled the C64 kernel is referred to as the kernal). Once memory management and assembly language were mastered, the power unleashed was considerable as the programmer obtained essentially complete control over the C64's circuitry.

For this reason, I still recommend programming the C64 with the use of an emulator, such as VICE or CCS64 running on a PC. I used to program the C64 when it was at its peak of popularity and recently returned to the C64 out of curiosity (and perhaps nostalgia). Re-learning what i had forgotten and taking it further I realised that such an exercise is valuable to programmers today - it gives one a deeper understanding of computer operation and good practice in bit/byte manipulation and hexadecimal-binary-decimal conversions. The old 6510 may be simpler than a modern PC, but it operates along very similar principles. Bit manipulation still has its place in modern high-level languages, but I became rusty with it, since it is used much less (most of the manipulations are automated and carried out underneath the covers) and returning to the C64 was good revision and practice in this skill. It is also interesting from a computing historical perspective, for example with the C64 we encounter pointers and automated garbage collection - forerunners to those used in C++ and C# respectively.

8-bit Multiplication

Another advantage of programming a C64 emulator, is that most of the programs written for this machine, and also many of the textbooks, are freely available for download online. There are a number of very good free books online explaining how to program the C64 in machine code using **assembly language**. Warning: many of these books were typed manuscripts, and since it is very easy to make careless errors in assembly code they often contain errors - to copy textbook programs one needs to have a thorough understanding of what the code is doing so that it may be debugged! Here we include a couple of debugged textbook examples. The assembly program below carries out 8-bit multiplication, that is it can multiply to byte-values together, i.e. it can multiply together two whole numbers, each in the range 0 to 255. This is harder than it sounds in assembly language, since the result has to be stored in two bytes, since the largest value that can be stored in one byte is 255.

Recall that computers store data in binary format, since they are made up of minute switches with each switch having only two states: on (value 1) and off (value 0). Each switch is thus a single **binary digit or bit**. Just as in usual base ten arithmetic, a single digit can take any one of the values: 0,1,2,3,4,5,6,7,8,9 in base two (binary) each digit (bit) can only assume one of the following values: 0, 1. Since old computers like the C64 were 8-bit computers, a byte or 'word' was 8-bits long and we still define a **byte** as 8-bits even in modern processors with 32 and 64 bit processors. In a 32 bit processor, for example, the **word** length is 32 bits or 4 bytes. A **nibble** (or nybble) is half a byte or 4

bits.

What do we mean by an '8-bit' processor?

A computer 'brain' consists in its simplest terms as the microprocessor or CPU (central processor unit, the 6510 processor in the C64, 6502 processor in the earlier Vic-20) and one or more memory banks, the two being connected by wires called buses. The address bus connects the CPU to the memory and enables the CPU to pick-out or select a memory location by its unique address. A **data bus** sends information from one part to another, such as the value retrieved from a memory location to the CPU. In the 8-bit architecture of the C64 each memory location stores one byte, that is it holds a number from 0 to 255 (negative numbers can also be represented but we will not discuss that hear - the interested reader should look-up 'two's-complement' and 'one's-complement'). The data bus is also b-bit, since it needs enough bandwidth to transfer 8-bit numbers around. The CPU also handles data as 8-bit bytes. The processor has its own memory locations as part of the chip, called **registers**. Registers are not referred to as 'memory' as they are separate from the RAM and ROM of the memory banks, but they are the equivalent in many ways to the working-memory of the human brain, both hold tiny amounts of data (7 chunks in the average human brain) for a short time only, storing data that is currently being worked upon. Each CPU register holds a single byte. One such register is the **accumulator** (A-register) which stores the result of the most recent arithmetic (which is carried out by part of the CPU called the **arithmetic and logic unit** or ALU). Two more key registers are the two **index registers**, the X and Y registers which are useful temporary stores in many arithmetical operations.

The **processor status register** (P-register) holds important information in its bits, which act as switches called flags. We typically number the bits in a byte from 0 to 7. Bit 0 of the status register is the **carry flag** (C-flag) and is set (made equal to 1 or switched on) whenever the accumulator overflows and rolls over from its maximum value of 255 to 0, as may happen if two numbers are totalled and their sum is greater than 255. Bit 1 is the **zero flag** (Z) and is set when the result of a computation by the ALU is zero. Bit 2 is the **interrupt mask** (I) and when set the computer ignores interrupts, such as the pauses necessary to scan the keyboard for user input. Bit 3 is the **decimal flag** (D) and when set then arithmetic occurs in binary coded decimal (BCD). Bit 4 is the break flag (B) and is set when a BRK (in assembly language) is executed, breaking from the current machine-code subroutine. Bit 5 is not used and is always set to 1. Bit six is the **overflow flag** (O or V) and is used in two's complement arithmetic, for dealing with negative numbers. Finally, bit 7 is the **negative flag** (N) or sign flag (S) and is set whenever a computation by the ALU produces a negative result (again this is two's complement arithmetic, in which the high bit (the one that normally has a value of 128) of the result is used to indicate a negative number (when set to one) or a positive number (when set to 0) - the N flag matches this high or most significant bit). Usually we deal only with the C, Z, O and N flags. In two's complement arithmetic an 8-bit word can store values ranging from -128 to 127.

Another register is the **stack pointer** (S). Instructions being processed are stored in a region of memory called the stack, because the rule is the first in. Last out, rather like a stack of plates (the last plate on is the first plate off) and this allows the computer to easily keep track of where it has got to in a program being executed, especially when the program has branching points or jumps to other subroutines. The stack pointer points to the address of the top of the stack. The **program counter** points to the address of the next instruction to be executed.

Although one 8-bit byte can only hold a number as large as 255 (2e8 - 1), two bytes can hold a number as large as (256 x 255) + 255 = 65535 (or alternatively 2e16 -1, the minus one allowing us to represent zero as 0000 0000 0000 0000). The C64 has 64 K (64 x 1024 bytes) of memory and each

location is numbered from 0 to 64 0000 odd. How then do we access a location higher than 255? This requires two bytes. **Memory pointers** in the C64 consisted of two bytes (they were 16 bit). The address bus contained two 8-bit channels, allowing 16 bits of data to be sent down it, so that all the available memory locations could be accessed by the CPU.

```
LDA IM 255
STA 900
LDA IN 200
STA 904
LDA IM 0
STA 902
STA 906
STA 907
LDY IM 8
LSR 900
BCC 19
LDA 906
CLC
ADC 904
STA 906
LDA 907
ADC 902
STA 907
ASL 904
ROL 902
DEY
BNE 223
LDA 906
STA 1025
LDX IM 1
STX 55297
LDA 907
STA 1024
STX 55296
RTS
```

```
A 2000 LDA #$FF          A9 FF
A 2002 STA $0384         8D 84 03
A 2005 LDA #$c8          A9 C8
A 2007 STA $0388         8D 88 03
A 200A LDA #$00          A9 00
A 200C STA $0386         8D 86 03
A 200F STA $038A         8D 8A 03
A 2012 STA $038B         8D 8B 03
A 2015 LDY #$08          A0 08
A 2017 LSR $0384         4E 84 03
A 201A BCC $202D         90 11
A 201C LDA $038A         AD 8A 03
A 201F CLC               18
A 2020 ADC $0388         6D 88 03
A 2023 STA $038A         8D 8A 03
A 2026 LDA $038B         8B 03
A 2029 ADC $0386         6D 86 03
```

```
A 202C STA $038B          8D 8B 03
A 202F ASL $0388          0E 88 03
A 2032 ROL $0386          2E 86 03
A 2035 DEY                88
A 2036 BNE $2017          D0 DF
A 2038 LDA $038A          AD 8A 03
A 203B STA $0401          8D 01 04
A 203E LDX #$01           A2 01
A 2040 STX $D801          8E 01 D8
A 2043 LDA $038B          AD 8B 03
A 2046 STA $0400          8D 00 04
A 2049 STX $D800          8E 00 D8
A 204C RTS                60
```

```
A9 FF 8D 84 03 A9 C8 8D
88 03 A9 00 8D 86 03 8D
8A 03 8D 8B 03 A0 08 4E
84 03 90 11 AD 8A 03 18
6D 88 03 8D 8A 03 AD 8B
03 6D 86 03 8D 8B 03 0E
88 03 2E 86 03 88 D0 DF
AD 8A 03 8D 01 04 A2 01
8E 01 D8 AD 8B 03 8D 00
04 8E 00 D8 60
```

```
169 255 141 132 3 169 200 141
136 3 169 0 141 134 3 141
138 3 141 139 3 160 8 78
132 3 144 19 173 138 3 24
109 136 3 141 138 3 173 139
3 109 134 3 141 139 3 14
136 3 46 134 3 136 208 223
173 138 3 141 1 4 162 1
142 1 216 173 139 3 141 0
4 142 0 216 96
```

Purely for interest, i have written out the binary below - this represents the actual machine code as seen by the computer hardware!

```
10101001  11111111  10001101  10000100  00000011  10101001  11001000  10001101
10001000  00000011  10101001  00000000  10001101  10000110  00000011  10001101
10001010  00000011  10001101  10001011  00000011  10100000  00001000  01001110
10000100  00000011  10010000  00010001  10101101  10001010  00000011  00011000
01101101  10001000  00000011  10001101  10001010  00000011  10101101  10101011
00000011  01101101  10000110  00000011  10001101  10001011  00000011  00001110
10001000  00000011  00101110  10000110  00000011  10001000  11010000  11011111
10101101  10001010  00000011  10001101  00000001  00000100  10100010  00000001
10001110  00000001  11011000  10101101  10001011  00000011  10001101  00000000
00000100  10001110  00000000  11011000  01100000
```

Machine Code Routine to Display a Byte Value as a Decimal

```
5 CLR
10 PRINT CHR$(147)
20 FOR D=0 to 73
30 READ Q: POKE 828+D,Q
40 NEXT D
50 SYS 828
100 DATA 162,1,169,###,201,200,144,15
110 DATA 233,200,72,169,50,141,0,4
120 DATA 142,0,216,104,76,100,3,24
130 DATA 201,100,144,12,233,100,72,169
140 DATA 49,141,0,4,142,0,216,104
150 DATA 24,160,0,201,10,144,5,200
160 DATA 233,10,201,10,176,249,72,152
170 DATA 105,48,141,1,4,142,1,216
180 DATA 104,105,48,141,2,4,142,2
190 DATA 216,96
```

(Replace ### with the value to be displayed as a decimal, with possible values ranging from 0 to 255).

This displays any one byte value from 0 to 255 in the top-left corner of the screen in decimal format. All that remains is to couple this routine with the first 8-bit multiplication routine so that the answer is translated for us. (I will do that when i get time!).

Let us examine the first multiplication subroutine for the example 255 x 2 = 510, using @ to mean 'memory address':

The maximum value of one byte (all 8 bits set = 1):

Recall that in a byte the least significant bit (the zero bit, usually written on the left) -

```
128  64  32  16  8  4  2  1
1    1   1   1   1  1  1  1
```

1111 1111(2) = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255 (10).

The maximum value of two bytes or a 16-bit word:

```
32768  16384 8192 4096  2048  1024  512  256 |  128  64  32  16  8  4  2  1
1         1      1     1      1     1    1     1      1    1   1   1  1  1  1  1
```

1111 1111 1111 1111 (2) = 32768 + 16384 + 8192 + 4096 + 2048 + 1024 + 512 + 256 + 255 = 65535 = 2e16 -1

The code on the left is the **assembly code** for our 8-bit multiplication subroutine; the values are given in decimal.

On the right is the code entered in the assembler, e.g. A 2000 LDA #$FF, followed by the assembled machine code in **hexadecimal**: A9 FF.

In hexadecimal (hex or base 16) each digit can take the values:
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 which are represented as:
0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
The $ indicates a hex number, so $FF is equivalent to (15 x 16) + 15 = 255.

Hexadecimal is used because each hex digit represents 4-bits or one nibble. FF is thus the max value that can be stored in one byte. Four hex digits can represent a 16-bit memory address.

The program written by hand is shown on the left and then assembled using the tfc3 C64 assembler (the cartridge image can be loaded into a C64 emulator). Each assembly language instruction is a three letter mnemonic, e.g. **LDA** 255 means load the accumulator (LDA) with the value 255, and places the value 255 into the A-register. Different **addressing modes** are used in machine language, but we will only use two here: **immediate addressing mode** does not access a memory location but instead places a value directly into a register, as we just did with LDA 255 which may be written LDA IM 255 (or LDAIM 255) to remind us that we are using immediate mode. This code also uses the **absolute addressing mode**, e.g. STA 1025 means store the value in the accumulator into address location 1025 (which is location $0401 in hex). Memory locations 1024 and 1025 are the first two character positions on the screen, in the top left corner, and is where the answer of the 8-bit multiplication is printed to the screen.

The first two numbers loaded into the accumulator, 255 and 200, are the two numbers we wish to multiply together - change these to any value between 0 and 255 for a different multiplication. Notice that immediate mode is indicated to the assembler using a # symbol, as in LDA #$FF. The hex digits in the rightmost column is the output of the assembler, and is the final machine code represented in hex. Notice that address locations and pointers to these locations are stored in memory in the reverse order to what we normally write (least significant byte before the most significant byte): e.g. A 2040 STX $0401 is stored as three bytes. In memory location $2040 is placed the byte 8E, the machine-code for the instruction STA, which represents a machine-code routine in ROM given the id number $8E (8 x 16 + 14 = 142). In location $2041 is the least significant byte of the address where the value in A is to be stored: $01, and in location $2042 is placed the value $04, representing the most significant byte of the address.

The subroutine ends with **RTS**, the instruction to return from the subroutine. We can run the final machine-code subroutine in the assembler software, or we can convert into decimal numbers and enter it using a BASIC program. The hexadecimal and its decimal equivalent are shown below:

These decimal values can then be loaded into memory using BASIC, as shown below:

```
5 CLR
10 PRINT CHR$(147)
20 FOR D=0 to 73
30 READ Q: POKE 8192+D,Q
40 NEXT D
50 SYS 8192
100 DATA 169,255,141,132,3,169,200,141
```

```
110 DATA 136,3,169,0,141,134,3,141
120 DATA 138,3,141,139,3,160,8,78
130 DATA 132,3,144,19,173,138,3,24
140 DATA 109,136,3,141,138,3,173,139
150 DATA 3,109,134,3,141,139,3,14
160 DATA 136,3,46,134,3,136,208,223
170 DATA 173,138,3,141,1,4,162,1
180 DATA 142,1,216,173,139,3,141,0
190 DATA 4,142,0,216,96
```

The machine-code in the DATA statements is read and poked into memory, starting at location 8192 ($2000). Once this is done, we can execute the machine code routine using the BASIC SYS command. Of course it takes time to load the routine into memory using BASIC, but once in memory we can call the subroutine as often as we like and benefit from the speed advantage of machine-code. PRINT CHR$(147) clears the screen, by the way.

If you run this program, then you will find that it prints two, often strange symbols in the top-left corner of the screen. If you use a C64 character screen code table, then you will find that the screen codes of these two characters give us the correct answer (most significant byte first). It is awkward having to do a manual translation! Another textbook assembly language program which i debugged as shown below:

One of the most fun features of assembly language is that it allows us to control the CPU's registers directly! When you enter the assembler program, also called a machine code monitor, you receive a display of the registers and their current values, something like this:

```
  PC      IRQ      BK      AC      XR      YR      SP      NV#BDIZC
 AB25    EA31     07      8D      00      0A      FP      *. **. . .*
```

Where: PC is the program counter, IRQ is the interrupt request, BK is the current memory bank, AC is the accumulator, XR the X-register, YR the Y-register, SP the stack pointer and NV#BDIZC the status register and its various flags (an asterisk indicates bit set = 1, a dot bit = 0). Now let's look at the code for the b-bit multiplication subroutine:

```
1.  LDA IM 2       load value 2 into accumulator
2.  STA 900        store A at address 900
3.  LDA IN 255     load value 255 into accumulator
4.  STA 904        store A at address 904
5.  LDA IM 0       load value 0 into accumulator
6.  STA 902        store A at address 902
7.  STA 906        store A at address 906
8.  STA 907        store a at address 907
9.  LDY IM 8       load value 8 into Y-register
10. LSR 900        bitwise shift right value in 900
11 .BCC 19         branch if carry (C) clear to line 19
12. LDA 906        load value at @906 into A
13. CLC            clear C, C = 0
14. ADC 904        add with carry the value at @904 to A
15. STA 906        store A at @906
16. LDA 907        load value at @907 into A
17. ADC 902        add with carry value at @902 to A
18. STA 907        store A at @907
```

19. ASL 904 arithmetic (bitwise) shift left value at @904
20. ROL 902 bitwise rotate left value at @902
21. DEY decrement Y
22. BNE 223 branch not equal (if Z = 0) to line 10

This part displays the result (on the screen, in white):

23. LDA 906 load the value at @906 into A
24. STA 1025 store value in A at @1025 (screen)
25. LDX IM 1 load X with value 1 (= color white)
26. STX 55297 store X in @55297 (color memory)
27. LDA 907 load the value at @907 into A
28. STA 1024 store value in A at @1024 (screen)
29. STX 55296 store X in @55296 (color memory)
30. RTS return from subroutine

Notes: Y acts as a loop counter, counting the number of bits remaining as we multiply by multiplying the multiplicand by each bit of the multiplier in turn. We do a similar thing in decimal, where we multiply by the first digit (the 1's) then by the second digit, the 10's etc. as shown below:

e.g.

```
  210
x  12
  420  (x 2)
+2100  (x 10)
 2520
```

Our algorithm is doing the binary equivalent!

Note the bitwise operations:

ASL (line 19) - shifts all the bits one place to the left, removing the leftmost bit to the carry flag, and replacing the space in the rightmost bit position with 0.

ROL (line 20) - shifts the bits left, but in rotation with the carry flag, C, removing the bit in the carry flag and placing this in the space at the rightmost bit and removing the leftmost bit and placing it in the carry flag.

Note also the conditional branch operations:

BCC branches if C = 0, i.e. if C is clear
BNE branches if and only if Z (zero flag) = 0, i.e. if Z is clear

Each branches a set number of bytes if the above condition is met, the number of bytes being given by the number following the instruction. This uses 2's complement: if the number is 127 or less than it branches that many bytes forwards, if the value is 128 or more, then this represents a

negative value and then the program branches 256-number bytes backwards. In practice, the 6510 processor requires the address of the instruction to be jumped to and this must be entered in assembly, e.g. BNE $2017. Another commonly used branch operation is JMP which jumps to a memory location unconditionally, e.g. JMP $2017.

Links

C64 wiki: http://www.c64-wiki.com/index.php/Main_Page

1. A = 2
2. @900 = 2 (10) = 00000010 (2)
3. A = 255
4. @904 = 255 (10) = 11111111 (2)
5. A = 0
6. @902 = 0
7. @906 = 0
8. @907 = 0
9. Y = 8
10. 900: 00000010 -> 00000001, C = 0
11. branches in this case to line 19
19. 904: 11111111 -> 11111110, C = 1
20. 902: 00000000 -> 00000001, C = 0
21. Y = 7, Z = 0
22. branch to line 10
10. 900: 00000001 -> 00000000, C = 1
11. does not branch in this case
12. A = 0
13. C = 0
14. A = 0 + 11111110 = 11111110
15. 906: 11111110
16. A = 0
17. A = A + 00000001 = 00000001
18. 907: 00000001
19. 904: 11111110 -> 11111100, C = 1
20. 902: 00000001 -> 00000011, C = 0

21. Y = 6, Z = 0
22. branch to line 10
10. 902: 00000000 -> 00000000, C = 0
11. branch to line 19
19. 904: 11111100 -> 11111000, C = 1
20. 902: 00000011 -> 00000111, C = 0

21. Y = 5, Z = 0
22. branch to line 10
10. 900: 00000000 -> 00000000, C = 0
11. branch to line 19
19. 904: 11111100 -> 11111000, C = 1
20. 902: 00000011 -> 00000111, C = 0

21. Y = 4, Z = 0
22. branch to line 10
10. 900: 00000000 -> 00000000, C = 0
11. branch to line 19
19. 904: 11111000 -> 11110000, C = 1
20. 902: 00000111 -> 00001111, C = 0

21. Y = 3, Z = 0
22. branch to line 10
10. 900: 00000000 -> 00000000, C = 0
11. branch to line 19
19. 904: 11110000 -> 11100000, C = 1
20. 902: 00001111 -> 00011111, C = 0

21. Y = 2, Z = 0
22. branch to line 10
10. 900: 00000000 -> 00000000, C = 0
11. branch to line 19
19. 904: 11100000 -> 11000000, C = 1
20. 902: 00011111 -> 00111111, C = 0

21. Y = 1, Z = 0
22. branch to line 10
10. 900: 00000000 -> 00000000, C = 0
11. branch to line 19
19. 904: 11000000 -> 10000000, C = 1
20. 902: 00111111 -> 01111111, C = 0

21. Y = 0, Z = 1
22. do not branch

Print result:
= value at @907 and value at @906
= 00000001 + 11111110
= (1 x 256) + 254 = 510

ANSWER = 510

Assembly Language Today

Assembly language is a **low level language**, meaning it is close to the operations of the chipset itself. Indeed if we go any lower than we arrive at binary machine code itself. Assembly and machine code must still be used when designing the architecture of CPUs and motherboards and graphics cards. Those who have studied microprocessor architecture, perhaps as part of a computer studies degree, will have already encountered it, though perhaps in a theoretical setting. The operating system (OS) kernel of the C64 (the 'kernal') consisted of machine code routines, stored in ROM, that can be called directly from BASIC or machine code. Modern systems contain multiple layers or shells, for example, a PC calls machine code instructions on power-up, called the BIOS. This then loads a command interface, such as DOS and finally the Windows 'operating system'. Users interact only with the windows, or occasionally DOS shells. Most programmers also interact with Windows, calling its component functions, often indirectly. for example, C# code may create a new window, but under the hood this executes several core routines in the Windows OS. This is one of the reasons why I recommend programming a C64 emulator, to gain a

first-hand feel for how the processor and graphics chips work. This older style programming is far more intimate!

Today, programmers might use low-level languages to program device drivers, which are generally written in C, which is quite a low level language, but not as low level as assembly. In C and also Java, for example, we can request that data be stored in CPU register, but this is only a request - the registers may be busy, since a modern CPU runs many softwares in the background which have to be protected from any standard program that is executing. (Registers are faster than accessing RAM, and if they can be used then certain operations can be sped-up). With the C64 we have considerable direct control over the CPU.

Several legacy computer systems used the 6510 processor, and machine-code written on one will run on others given certain modifications, e.g. accounting for different arrangements of computer memory. Assembly is assembled into machine code before execution. This is similar to the way **compiled languages** like C work - a compiler translates the C code into machine language, specific to the given machine. Compilers, however, do not simply translate code, they attempt to optimise it and so are more complex than assemblers. Java and C# are a bit different. Both are **high-level langauges** and are compiled into an intermediate low-level language which is then translated in sections, as needed, by just-in-time (JIT) compilers into machine code during program execution. Java, for example, compiles into byte code (virtual machine code) which is an **interpreted language**. An interpreter translate code, line by line, into machine code during execution. However, the JIT can compile sections of the code into machine code, so that if the code is re-used no further interpretation is needed.

On the C64, BASIC was an interpreted language, and so runs comparatively slowly as the interpreter slows down code execution. This was one of the main reasons why most games written for the C64 were written in assembly and assembled into machine code before execution - this made the machine code much, much faster than BASIC! (Often machine code is too fast, even on these old processors, and must be slowed, for example when animating graphics). Even without an assembler, it was possible to write assembly code and translate into machine code manually, using look-up tables to obtain the decimal values which could be used in a BASIC program. This approach is, of course, very error prone. Assembly language is relatively easy to learn but very hard to use! It took us 30 lines to multiply two 8-bit numbers together, and a few more to output the answer in human language! Clearly, this approach would be madness for writing a large windows application! That is why we use higher level languages like Java and C#! However, assembly still has its users to those who design system architectures and learning to use it on a C64 is a very rewarding experience!

Comment on this article!