nesdoug

26. ASM Basics

Intro to 6502 ASM, with ca65.

I've had a lot of questions about 6502 ASM. One of the features of cc65 is the ca65 assembler, which is a very good one. You can write any, or all, your functions in assembly. But, it would help if you knew how 6502 ASM works…so I'm going to write a few tutorials. All my examples will be ca65 specific. It would also help, if you have a strong understanding of binary and hexadecimal numbers.

The NES has 256 zeropage RAM addresses (that is addresses 0 – 0xff) and 1792 non-zeropage RAM addresses (addresses 0x100 – 0x7ff). Some games also have an additional RAM chip on the cartridge, usually mapped to 0x6000-0x7fff. If powered by a battery, it SAVES the game.

The hardware stack goes from 0x100-0x1ff, so let's not put any variables here. Also, most games put a OAM (sprite) buffer from 0x200-0x2ff. And, furthermore, if you are using cc65, it needs to put its stack and (optionally) heap somewhere…usually at the top (around 0x7ff)…but you can change this in the .cfg file.

Anyway, with these assumptions…I'm going to use the zeropage (0 – 0xff), and 0x300-0x3ff in all the examples. I will usually use $ to indicate hex numbers, rather than 0x. This is requirred syntax by ca65.

Declaring constants.

```
zip = 0
FOO = $3f
FOO2 = $03e3
```

When used in assembly code, the assembler will replace the symbol with the value you define.

Examples:

-> here means 'assembles into'
the left-most part is what you will be typing

```
LDA #zip -> LDA #0   load A with value 0, the '#' means value, not address
LDA zip  -> LDA 0  load A from the 8-bit address $00
LDA #FOO -> LDA #$3f load A with the value $3f
LDA FOO  -> LDA $3f  load A from the 8-bit address $3f
LDA FOO2 -> LDA $03e3 load A from the 16-bit address $03e3
LDA #<FOO2 -> LDA #$e3 load A with the value (lower byte of $03e3 = $e3)
LDA #>FOO2 -> LDA #$03 load A with the value (upper byte of $03e3 = $03)
```

< gets the lower byte of a 2 byte expression
> gets the upper byte of a 2 byte expression

LDA #FOO2 -> produces an error. The assembler was expecting an 8-bit number.

Declaring variables.

```
.segment "ZEROPAGE"
foo: .res 1
bar: .res 2
```

Assuming this is the first thing the assembler sees… foo will be reserved 1 byte at address $00, and bar will be reserved 2 bytes at addresses $01 and $02.

```
LDA foo  -> LDA $00 load A from the 8-bit address $00
LDA bar  -> LDA $01 load A from the 8-bit address $01
LDA bar+1  -> LDA $02 load A from the 8-bit address ($01 + 1 = $02)
```

```
.segment  "BSS"
fooz: .res 1
baz: .res 2
```

As I described above, I'm defining the BSS section in the .cfg file as being from $300-$3ff. Therefore, the assembler will reserve 1 byte for fooz at $0300 and 2 bytes for baz at $0301 and $0302.

```
LDA fooz  -> LDA $0300 load A from the 16-bit address $0300
LDA baz  -> LDA $0301 load A from the 16-bit address $0301
LDA baz+1  -> LDA $0302 load A from the 16-bit address ($0301 + 1 = $0302)
```

Importantly, we don't need to know what value is reserved when writing code. The assembler will keep track of the values and addresses of every label, you just have to reference them in your code using the labels/variable name.
* constants and variables should be defined at the very top of the ASM page.

Referencing ROM addresses in code, using labels.

```
.segment  "CODE"
LDA Table1
LDA Table1+1
...
Table1:
.byte $01, $02
```

(note, you could also put Table1 in the "RODATA" segment, if you like)

Let's say that the assembler has calculated that Table1 will be at address $8050. This will assemble into…

```
LDA $8050 load A from the 16-bit address $8050
LDA $8051 load A from the 16-bit address $8051
```

when the program is RUNNING, the first line will load A with the value #01 and the second line will load A with the value #02…because those are the values in the ROM at 8050 and 8051.

OK, now that we understand how the labels work, let's do some code…using C examples, and how to do it in ASM. There are 3 registers in the 6502. A, X, and Y.

foo = 3;

```
LDA #3          load A with value 3
STA foo         store A at address foo
```

or we could have used the other registers…

```
LDX #3          load X with value 3
STX foo         store X at address foo
```

or

```
LDY #3          load Y with value 3
STY foo         store Y at address foo
```

There is no difference which register you use for this kind of thing.
bar = $31f; //a 16-bit value

From working with cc65, I now have a habit of using A for low bytes and X for high bytes (as the cc65 compiler tends to do)…

```
LDA #$1f        load A with the value $1f
LDX #$03        load X with the value 3
STA bar         store A to address bar
STX bar+1       store X to the address bar+1
```

```
we could also have done...
LDA #$1f        load A with the value $1f
STA bar         store A to address bar
LDA #$03        load A with the value 3
sta bar+1       store A to the address bar+1
```

baz = bar; //a 16-bit value

```
LDA bar         load A from the address bar
LDX bar+1       load X from the address bar+1
STA baz         store A to the addres baz
STX baz+1       store X to the address baz+1
```

```
again, we could have done...
LDA bar         load A from the address bar
STA baz         store A to the addres baz
LDA bar+1       load A from the address bar+1
STA baz+1       store A to the addres baz+1
```

Next thing…increment / decrement

++foo;

```
INC foo  add 1 to the value stored at foo
```

– -foo;

```
DEC foo  subtract 1 from the value stored at foo
```

you can also increment and decrement the X and Y registers

```
INX  add 1 to the X register


INY  add 1 to the Y register


DEX  subtract 1 from the X register


DEY  subtract 1 from the Y register
```

You will have to use adding/subtraction to ++ or – – the A register.
Which brings us to simple math…in fact very very simple math. The 6502 can only do addition and subtraction, and bit-shift multiplication. And, ONLY the A register can do math or bit-shifting.

Adding is always done 'with carry'. The 6502 has certain FLAGS to assist math, and for doing comparisons. If the result of addition is > 255, then it sets the carry flag – in case you are doing 16-bit math (or more). If the result of addition is <= 255, the the carry flag is reset to zero. But, it always adds A + value + carry flag. Therefore, we must 'clear the carry flag' before addition. Here's an example…

```
A reg. + value + carry flag =
result now in A // carry flag
4+4+0  = A = 8, carry = 0
4+4+1  = A = 9, carry = 0
255+4+0  = A = 3, carry = 1
255+4+1 = A = 4, carry = 1
```

foo = fooz + 1; //8-bit only

```
LDA fooz        load A from address fooz
CLC             clear the carry flag
ADC #1          add w carry A + value 1, the result is now in A
STA foo         store A to the address foo
```

or, reverse them, get the same result…
foo = 1 + fooz; //8-bit only

```
LDA #1          load A with value 1
CLC             clear the carry flag
ADC fooz        add w carry A + value at address fooz, result now in A
STA foo         store A to the address foo
```

Let's do a 16-bit example.

bar = baz + $315; 16-bit values

```
LDA baz         load A from address baz
CLC             clear the carry flag
ADC #$15        add w carry A + value $15 (the low byte)
  if the result is > 255, the carry flag will be set, else reset to zero
STA bar         store A to the address bar
LDA baz+1       load A from the address baz+1
  ...notice, we don't clear the carry flag, we are using the
  carry flag result of the previous addition as part of this addition
ADC #$03        add w carry A + value $03 (the high byte)
STA bar+1       store A to the address bar+1
```

And, some **subtraction**. Like the ADC, subtraction always uses the carry flag, but in reverse. It's called Subtract with Carry. You need to SET the carry flag before a SBC operation. If the result of subtraction underflows below 0, it will reset the carry flag to zero. Else, it will set the carry flag. Again, this is in case you want to do 16-bit (or more) math.

Here's some examples…
! = NOT…ie, the opposite

```
A reg. - value - !carry flag =
result now in A // carry flag
8-4-!1  = A = 4,  carry = 1
8-4-!0  = A = 3, carry = 1
4-5-!1  = A = 255, carry = 0
4-5-!0  = A = 254, carry = 0
```

foo = fooz – 1; //8-bit only

```
LDA fooz        load A from address fooz
SEC             set the carry flag
SBC #1          subtract value 1 from A, result is now in A
STA foo         store A to address foo
```

```
And the reverse, which is a different thing altogether...
foo = 1 - fooz; //8-bit only
```

```
LDA #1          load A with value 1
SEC             set the carry flag
SBC fooz        subtract value at address fooz from A, result is now in A
STA foo         store A to address foo
```

```
And a 16-bit example...
bar = baz - $315; //16-bit numbers
```

```
LDA baz         load A from address baz
SEC             set the carry flag
SBC #$15        subtract value $15 from A, result is now in A
STA bar         store A to address bar
LDA baz+1       load A from addres baz+1
  ...notice, we DON'T set the carry flag. We are using the result
  of the last math to set/reset the carry flag.
SBC #$03        subtract value 3 from A (and subtract !carry), result now in A
STA bar+1       store A to the address bar+1
```

Stay tuned for many more ASM lessons to come.

March 10, 2016April 13, 2017 dougfraker