Whatever you like - Coming to Addressing Modes

dustlayer.com/cpu-6510-articles/2013/5/23/whatever-you-like-coming-to-addressing-modes

Telling the CPU what you want

The 6510 is in a way asking for explicitly. Let's take the command **LDA** which loads a Byte into our Accumulator Register. Whether you want to use LDA to read the content of a memory address or maybe directly a specific Integer Number or use the operand passed to LDA as a vector to point to another location in memory are all different instructions yet we use the same keyword in our assembler code - LDA.

For the beginner this can be mind-twisting because you can achieve different things with the same three-letter assembler code. So what is important to understand is not not only what the actual keyword technically does but how we ask the keyword to execute in a specific way. Depending on that, the CPU will take different routes and internally encode our keyword internally into a different Byte value.

The 6510 knows **13 Addressing Modes** and most of them are actually straight-forward. The point of choosing the correct addressing mode is to deal with locations or values in RAM and ROM where we usually are concerned about arranging Bits and Bytes around. All CPU instructions are either one, two or three Bytes in length.

Single-Byte Instructions cannot reference an address or value, they are explicit in themselves. A command like TXA transfers the content of the X-Register into the Accumulator - you cannot do anything different with the command. **Two-Byte Instructions** come with an additional Byte which can be treated as a plain value or Memory Location depending on which Addressing Mode we use. Finally **Three-Byte Instructions** are not much different their Two-Byte long sibling but come with two Bytes of Data, e.g. to load and store 16-Bit Addresses. Remember the C64 is an 8-Bit Machine but has a 16-Bit Address Bus otherwise we would not be able to reach any location outside $FF.

So all those 6510 instructions can be used with one or more addressing modes to further specify what we actually want to achieve. Let's run down the different modes with examples.

13 Paths to Success - The 6510 Addressing Modes

Accumulator Addressing correspond to all commands where the Accumulator is implied as operand, so no address needs to be specified

```
lsr ; do a logical shift right on Accumulator content
lsr A ; alternative syntax but operand A is not required
```

Implied Addressing is actually not a real addressing mode but the destination is already implied within the command. For example **Return from Subroutine** (**RTS)** is a command which does not come with any further information - it is implicit.

```
tsx ; saves current Status Register into X-Register via Implied Addressing
rts ; returns from subroutine via Implied addressing
```

Immediate Addressing s used to work with a value rather than a memory location, e.g. if you want to load the decimal value 15 into the accumulator you would use LDA #$0F . Immediate Addressing can be identified by the Hashpount (#). Whenever you use this Prefix you always deal with the actual value but not with a Memory Location.

```
lda #$40 ; load Accumulator with the Value #$40 (Decimal 64)
```

Relative Addressing is a mode used by Branch instructions like BNE, BEQ etc. They work with a signed 8-Bit long offset, that means that Bit#7 is used to indiciate whether the offset value is negative or positive. When Bit#7 is set, the value is considered to be negative. In your day-to-day programming you often use Branch instructions with labels so the assembler takes care of calculating the offset to the target label of your branch instruction. Just keep in mind that the maximum range you can branch in your code is 127 instructions forward or 128 backwards. Branch instructions are faster than the JMP instruction so avoid using JMP in favor of Branching if possible.

```
; Example 1

ldx #$00 ; load X Register via immediate addressing
move_ship dec $d000 ; decrease x-Coordinate of Sprite#0 via absolute addressing
inx ; increase X Register via implied addressing
bne move_ship ; branch to move_ship via relative addressing if X is not zero

; Example 2
lda #$f9 ; specify a Raster Line (249) via immediate Addressing
cmp $d012 ; check $d012 if Raster Line has been reached yet
bne *-3 ; branch back 3 bytes until Raster Line matches via relative addressing
```

Absolute Addressing use a full 16-Bit address to reference to a target memory location.

```
lda $d020 ; load current border color via absolute addressing
sta $d021 ; store color information in background color register via absolute addressing
```

Absolute Indexed by X Addressing means that the address in question is made up from the 16-Bit location plus the content of the X Register. The instruction LDA $0400,x will load the value from the location $0405 when the X Register holds the Value #$05.

Absolute Indexed by Y Addressing is the exact same as the mode above but used in conjunction with the Y Register.

```
;Absolute Indexed by X Register example
;Absolute Indexed by Y Register is analogous

ldx #$00 ; load X Register via immediate Addressing
lda #$02 ; load Accumulator via immediate Addressing
loop sta $d800,x ; store value for color red into Color RAM for $400+x via Absolute Indexed by X Addressing
```

```
inx ; increase X via implied Addressing
bne loop ; repeat until 256 cells have been colorized via Relative Addressing
```

Absolute Indirect Addressing is only used by the **JMP** instruction and it may sound a bit weird at first. What it does is to jump to a specified 16-Bit memory Location and considers the found value in that location to be the least significant Byte of another 16-Bit address which is combined with the following Byte to make up the real target address of the Jump instruction. For example the Reset vector **JMP ($FFFC)** first jumps to a location in the standardized Commodore Jump Table which on the C64 has the value $E2 followed by the value $FC. This address is then used for the actual Jump. The C64 Reset Routine is therefor located at **$FCE2 (64738)** which might be different on other Commodore Machines. So when you develop cross-platform, say for VIC-20 and C64 you can use **JMP ($FFFC)** on both machines though the actual location of the routine differs. That saved some time when you did conversion jobs in the past across Commodore Computers.

```
; assume that $0801 must hold a specifc value for copy protection reasons
; if the value is not in there we want to reset the C64

reset_vector = $fffc
run_game = $c000

lda $0801 ; load content of $0801
cmp #$50 ; conpare with value #$50
beq continue ; if successful branch to continue label via Relative Addressing
jmp (reset_vector) ; jump to reset routine via Absolute Indirect Addressing
continue jmp run_game ; jump to game routine via Absolute Addressing
```

Zeropage Addressing is actually not a real addressing mode of the Chip per se. It is basically the same as the Absolute Addressing Mode but works with the Zeropage. The Zeropage (Page 0) covers Memory Locations $00 - $FF. The cool thing about using Zeropage Addressing is that the Chip ignores the most significant Byte as it is zero anyways so this saves a Byte which shortens programs and increases execution speed. That's also the reason why the 256 Bytes of the Zeropage are usually in great demand of C64 programs. If you use BASIC in addition to your Machine Language routines it is crucial to consider that lots of Zeropage addresses are already used by the BASIC Interpreter. Zeropage Addressing works likewise as Absolute Addressing so **Zeropage Indexed by X**, **Zeropage Indexed by Y** and of course just the Standard **Zeropage Adressing** in the fashion **LDA $35** are possible.

Zeropage Indirect Indexed Addressing is a very popular solution to address any location in the C64 memory with just two Bytes. The idea is that you use two locations in Zeropage to store a 16-Bit address - as usual in low/high order. You put the location holding the least significant Byte in parentheses and use it in conjunction with the Y register to ultimately address any location in C64 Memory.

Let's say we want to clear the screen using a a loop by using Zeropage Indirect Indexed Addressing. For that we store the value **$00 and $04** in **$FB/$FC** since our Screen RAM start at $0400. We can now use the command **LDA ($FB),Y** to address the first 256 locations from $0400 to $04FF with a two-byte instruction. Once Y becomes zero again, all we need to do is to increment the value in $FC and start over to address $0500 to $05FF and so forth. Here comes the example code.

```
; When Addressing Modes use 8-Bit in favour of 16-Bit encoded locations
; they are commonly called Zeropage addressing modes

; Example: Zeropage Indirect Indexed Addressing
; Fill the whole screen with Spaces.

ldx #$04 ; load X-Register via Immediate Addressing
sta $fb ; store into $fb via Zeropage Addressing
lda #$04 ; load Accumulator via Immediate Addressing
sta $fc ; store into $fc via Zeropage Addressing
lda #$20 ; load Y-Register Spacebar code via Immediate Addressing

loop sta ($fb),y ; store Accumulator value by Zeropage Indirect Indexed Addressing
iny ; increase Y by implied Addressing
bne loop ; branch to loop until 255 locations have been addressed via Relative Addressing
inc $fc ; Increase most significant Byte to point to next page in Screen RAM
dex ; decrease X-Register by implied Addressing
bne loop ; branch to Loop to continue filling the screen with Spaces via Relative Addressing
rts ; after four rounds we are done and return
```

Zeropage Indexed Indirect Addressing is a rarely used mode. the vector is this time chosen by adding the value in the X Register to the given 16-Bit Zero Page address. For example when the X-Register holds the value #$01 then **LDA ($FB,X)** will address the location it finds in **$FC/FD** since X is added to the least significant portion of the actual 16-Bit address before the vector is used. Again this mode is not often used.

```
; Example: The value $02 in X is added to $15 for a sum of $17.
; The address $D010 at addresses $0017 and $0018 will be where the value $0F in the accumulator is stored.

ldx #$02 ; load X-Register via Immediate Addressing
lda #$05 ; load Accumulator via Immediate Addressing
sta ($15,x) ; store via Indexed Indirect that is to location $20
```

Last Hint: A common source of failure

My personal most popular mistake is to forget the Hashpound ("#") when I want to do Immediate Addressing. This is a mistake that is not always immediately (ha!) obvious when you run the code because whether you do **LDA $01** or **LDA #$01** might not make a visible difference at first when the content of the memory location $01 actually has the value One at the time of execution. This in particular occurs to me when I use symbols, so remember **LDA delay_counter** loads the value by the means of Absolute Addressing while **LDA #delay_counter** on the other hand loads just the plain value referenced via Immediate Addressing.

If something does not seem to make sense when loading and storing values, first check if you really do immediate addressing where you planned it to do.

-act